Product Identification

Product Name:	MODCAL
Product Mnemonic:	MODCAL
Project Number:	6651-0064
Project Engineers:	Steve Saunders Chris Maitz Ron Smith Jon Henderson
Project Manager:	Jean Danver
Product Manager:	Ann Tse
Product Assurance Engineer:	Ray Ling

Product Abstract:

MODCAL is the systems programming language for the Vision Computer Family (VCF). It is based on HP Standard Pascal. Features have been added to enable high level systems programming. MODCAL will be implemented on two architectures in the Business Computer Group--Vision and the HP3000. The HP3000 version will be used to write systems software, primarily in the data-base and data comunications area that will be ported to Vision.

Table of Contents

Version 6.3 Changes

Version 6.3 of the MODCAL ERS contains a number of changes to content and formatting. A summary of the differences is given below:

- Corrected Module Qualification syntax.
- Marked several features as NOT IMPLEMENTED.
- Corrected assignment mechanism for Procedural variables.
- Extended the definition of *Sizeof*.
- Made a separate section for READONLY.
- Added a section for ANYVAR.
- Added Object/Offset model diagram.
- Added an example of *XCall*.
- Updated predefined assembly instruction for Vision foundation module.
- Added $PRIVATE_PROCE OFF$ requirement to OPTION Inline.
- Added OPTION Uncheckable_ANYVAR.
- Modified OPTION Extensible_Gateway to work on the 3000.
- Added tables of permitted Definition Option / Directive combinations.
- Added $SET, $IF, $ELSE, and $ENDIF compiler options.
- Added several missing Pascal/3000 compiler options.
- Expanded the descriptions of $PUSH$ and POP.
- Added new reserved word ANYVAR
- Changes are shown in a **BOLDER** type face.
- Features that will not be implemented soon are shown in a smaller type face.

Version 6.1 Changes

Version 6.1 of the MODCAL ERS contains the XCall Mechanism.

Version 6.0 Changes

Version 6.0 of the MODCAL ERS contains the changes made that were reflected in the Delta ERS (May 19). A summary of the differences is given below:

- Try recover statememt and Escape function are not permitted in code that will execute on the ICS.

- Variable type coercion is permitted only for actual parameters

- The grammar for object/offsets was changed.

- Predefined procedure *BuildPointer* was added

- Predefined function *CurrentObject* was added.

- Predefined function *GetObjectSize* was added.

- Predefined procedure TestDown was changed to a Boolean function.

- Procedure definition options (Gateway, Extensible_Gateway, Inline, Default_Parms, Interrupt_Parms and Unresolved) were added.

- Predefined functions HaveExtension and HaveOptVarParm were added.

- The size of crunched structures is temporarily limited to 2048 words (32 k-bits) or less.

Version 5.0 Changes

Version 5.0 of the MODCAL ERS contains many significant changes and additions over versions 3.0 and 4.0 (both of these versions were very similar in content). The differences in the order they appear in version 5.0 are:

- The syntax notation was changed to use regular expressions.

- Definition module feature was added.

- Export qualifiers were added.

- *Escape* and *EscapeCode* definitions were refined.

- Type coercion definition was rewritten.

- Variable type coercion was added.

- *ReOrd* function was added.

- Procedural type and *Call* definitions were refined.

- Functional type and *FCall* definitions were added.

- *Sizeof* and *Addr* functions were added.

- Schema array feature definitions were refined.

- Withobject syntax was extended.

- Predefined object procedure definitions were refined.

- *Stack* function was added.

- Predefined pointer arithmetic procedure/function definitions were refined.

- *OffsetPart* and *ObjectPart* functions were added.

- Crunch packing definition was rewritten.

- *Move_R_to_L* and *Move_L_to_R* procedures were added.

- Multiple heaps definition was removed.

- Portability foundation definitions were refined.

- Assembly language predefined procedures/functions were added.

- Definition option explanations were added.

- Unresolved definition option was added.

- Inline and Split directive stuff was deleted.

- Directive explanations were refined.

- Compiler option explanations were added.

- Data type / data representation matrix was added.

- Data representation explanations were added.

- Type compatibility definitions were refined.

- Reserved word lists were added.

Introduction

The original Pascal programming language is described in the Pascal User Manual and Report by Jensen and Wirth. Three of the principal design goals of the Pascal language were:

1) The language should be suitable for teaching the concepts of structured programming,

2) It should be possible to construct simple, efficient and portable compilers for the language, and,

3) The language should be type-safe and allow compile-time type checking.

These design goals were essentially met; however they resulted in Pascal being based on a very simple, restricted machine model. The Pascal virtual machine consists of a program code area, a stack area for local and global variables, and a heap area for dynamically allocated data. This machine model does not give the programmer access to the full power of the underlying computer architecture. For example, within Pascal there is no way of storing and accessing variables in HP3000 extra data segments or Vision objects.

This is not surprising, since Pascal was never intended to be a systems programming language. It is, however, quite feasible to use Pascal as a basis for a systems programming language. Mesa and Ada are two prime examples of Pascal-based systems programming languages. The main sacrifice in converting Pascal to systems programming use is the loss of full type-safety.

This document describes a set of extensions to HP Standard Pascal that enables it to be useful for systems programming work such as operating systems, data bases, and data communications. This extended version of Pascal is called MODCAL.

MODCAL is an acronym for Modular Pascal. Hence, one of its main features is a module facility, allowing for separate compilation, encapsulation, and libraries.

In MODCAL heavy emphasis is placed on facilitating the construction of machine-independent programs, while not disallowing access to low-level machine features where necessary. This is done through such features as type coercion (interpreting data as a different type than it is declared), *Sizeof* and *Addr* functions and procedure variables.

A Hewlett Packard Corporate Standard exists for MODCAL. This document is organized to serve as a basis for the corporate standard. The table of contents reflect the main elements of this standard, required, adaptable, nonadaptable and experimental.

Required is a list of extensions to HP Standard Pascal which must be in all Modcals in the coporation. Adaptable are a list of extensions which appear in some existing Modcals which may not be possible or advisable to put in another. However, the extension is easily adapted to another extension in the other MODCAL. Nonadaptable are extensions to HP Standard Pascal which are in an existing MODCAL today, but is so machine dependent that using it would make porting very difficult. Experimental features are those agreed to be desirable. However, they have not yet been implemented or tested by an existing MODCAL.

BCG MODCAL for the HP3000 and VCF contains some experimental features designed to allow the development of system code on the HP3000 that is of acceptable efficiency and transportable to VCF. There is support for the notion of an "object", which is found on HP3000, VCF, and Dawn architectures. Crunch packing allows data structures to be defined which will have the same bit offsets, no matter what computer the data was transfered to. Schema Array types was designed to allow a type safe way of passing, allocating and slicing different sized arrays.

Notation

The syntax of the extensions are described by using a grammar notation with regular expressions.

Terminal symbols are distinguished from other symbols by surrounding quotes, e.g. 'if'. Nonterminals are represented by standard identifiers (a letter followed by zero or more letters, digits, or underscores) enclosed in angle brackets ('< >'). The following are other meta symbols used in the notation:

->	means the nonterminal on the left is replaced with the sequence of symbols on the right
*	means zero or more occurrence of what precedes it
+	means one or more occurrence of what precedes it.
?	means optional, zero or one occurrence of what precedes it.
|	means alternative, either an occurrence of what precedes it or an occurrence of what follows it.
list	means one or more occurrence of what precedes it separated by an occurrence of what follows it
()	parentheses to change binding rules
(juxtaposition)	means concatenation

Operator precedence hierarchy

()	most binding
?,*,+,list	next, left to right
(juxtaposition)	next
|	least binding

Examples:

(a typical expression grammar)

```
E -> E '+' T        F -> 'i'
  -> T

T -> T '*' F
  -> F
```

(The productions for a given nonterminal are collected together and the left part is factored out.)

```
A -> 'a'  'b'*
   a, ab, abb, abbb, abbbb, ...

A -> 'a'  'b'+  'c'
   abc, abbc, abbbc, abbbbc, ...

A -> 'a'  'bc' list '$'
   abc, abc$bc, abc$bc$bc, ...

A -> 'a' ('b' | 'c')? 'd'
   ad, abd, acd
```

A. HP MODCAL Standardized and Required Features

1. HP Standard Pascal

At the core of the MODCAL language is HP Standard Pascal. It makes up almost 90% of the language definition. HP Standard Pascal is not described in this document. A description can be found in the HP Standard Pascal document or the Pascal/3000 Reference Manual.

2. Modules

Note: Implementation of this feature requires segmenter changes. The extensions to Modules from HP Standard Pascal (Implement, Definition, Readonly, Hidden, Qualified) have been recently defined and are not yet implemented in any MODCAL. These features are subject to modification, especially if implementation difficulties arise.

A module is a program fragment which can be compiled independently and later used to complete otherwise incomplete programs. A module usually defines some constants, data types and variables, and procedures which operate on these data. Such definitions are made accessible to users of the module by its export declarations. The module may itself make use of other module definitions by importing the other modules.

Additional Reserved Words

MODULE | EXPORT | IMPORT | IMPLEMENT | DEFINITION |
READONLY | HIDDEN | QUALIFIED

a. Programs and Modules

The source text input to a compiler (complete unit of compilation) may be a program or a list of modules. The input text is terminated by a period.

```
<compilation_unit>
        ->  <program> '.'
        ->  ( <module> list ';' )  '.'
```

A module is a program fragment which can be compiled independently and later used to complete otherwise incomplete programs. An implementation may allow only a single module of input at a time, requiring multiple invocations of the compiler to process several modules. It will probably be implemented this way in MODCAL/3000 and MODCAL/VCF cross compiler.

b. Programs

A MODCAL program is in form very similar to a procedure declaration. It differs in its heading, and in that modules may be declared and imported in the declaration part of the program's block.

```
<program>
    ->  <program_heading> <program_block>
```

```
<program_heading>
   -> 'PROGRAM' <identifier>
                     ( '(' <program_parameters> ')'  )? ';'

<program_parameters>
   -> <identifier> list ','
```

The identifier following the symbol program is the program name; it has no further significance inside the program. The program parameters denote entities that exist outside the program and through which the program communicates with its environment.

These external entities (usually files) must be declared in the block which constitutes the program like ordinary local variables.

Identifiers declared to be program parameters may be of any Pascal type. External entities which are not files are implementation defined.

The two standard files Input and Output must not be declared but have to be listed as parameters in the program heading, if they are used. The initializing statements *Reset* (Input) and *Rewrite*(Output) are automatically generated and need not be specified by the programmer.

Examples

```
PROGRAM copy (f,g);
VAR f,g: FILE OF real;
BEGIN
Reset(f);
Rewrite(g);
WHILE NOT Eof(f) DO
   BEGIN
   g^ := f^;
   Put(g);
   Get(f)
   END;
END.
```

```
PROGRAM copytext(input,output);
VAR ch:char;
BEGIN
   WHILE NOT Eof(input) DO
      BEGIN
      WHILE NOT Eoln(input) DO
         BEGIN
         Read(ch);
         Write(ch);
         END;
      Readln; Writeln
      END;
END.
```

c. The Program Block

Identifiers declared in the <program_block> are said to have global scope; that is, they constitute the outermost level of the program's declaration structure.

```
<program_block>
  ->  <label_declaration_part> <global_declaration>*
        <procedure_and_function_declaration_part> <statement_part>

<global_declaration>
      ->  <constant_definition_part>
      ->  <type_definition_part>
      ->  <variable_declaration_part>
      ->  <module_declaration>
      ->  <import_part>

<module_declaration>
      ->  <module> ';'

<import_part>
      ->  'IMPORT'  ( <module_identifier> list ',' )  ';'

<module_identifier>
      ->  <identifier>
```

Note: <label_declaration_part>, <procedure_and_function_declaration-part>, <statement_part>, <constant_defintion_part>, <type-defintion_part>, <variable_declaration_part>, <identifier> are defined as in HP Standard Pascal.

Program blocks, unlike the block of a procedure or function, may contain module declarations. A module is a collection of global declarations which may be compiled independently and later made part of a program block. Any module used by a program, whether appearing in the program's globals or compiled separately, must be named in an import list. Modules, and the entities they export, always belong to the global scope of a program which uses them.

A module cannot be imported before its definition has been compiled, either as a global of the importing program or by a previous invocation of the compiler. This limits construction of mutually-referring modules. Access to separately compiled modules is discussed in Section 1e.

d. Modules

Although a module declaration defines data and procedures which will become globals of any program importing the module, not everything declared in the module becomes known to an importer. A module specifies exactly what will be exported to the "outside world", and lists any other modules on which the module being declared is itself dependent. That is, a module has access only to those global enitities which it either declares itself or imports. The only exception is the program parameters INPUT and OUTPUT. The default values for predefined file procedures and functions are allowed. An error will be issued, usually by the linker/segmenter if INPUT or OUTPUT are used by a module and are not program parameters of the importing program.

```
<module>
    ->  'MODULE'  <module_identifier>  ';'
             <import_part>? <export_part>*
             <implement_part>  'END'
    ->  'DEFINITION' 'MODULE' <module_identifier>  ';'
           <import_part>?  <export_part>+  'END'
```

There are two ways to define a module. A module can be completely defined at once using the first form above. A module defined this way must have an <export_part>. A module which did not export anything would be inaccessable, hence useless. A module can be defined in two parts, allowing for a certain amount of mutual referral of modules. This is done by using the 'DEFINITION' 'MODULE' declaration. Here, anything the module will export is defined. Hence the <export_part> is required as in the complete definition. Any modules needed for the export are listed in the <import_part>. There is no <implement_part>. In order for the module to be completely defined, a module definition must appear later in the program with an implement section. A warning will be issued for each incompletely defined module (has DEFINITION MODULE only) in a <compilation_unit>. This definition must not have an <export_part>, but it may have an <import_part>. (Allowing this second <import_part> has not been definitely decided. It is desirable because it allows mutually referring procedures in different modules. It also cuts down the amount of information loaded into symbol tables at compile time of the importing module. It is undesirable because the 'Definition' 'Module' would not contain all the dependencies of that module.).

The <import_part>, as in a program block, is a list of module identifiers. The <export_part> defines constants and types, declares variables, and gives the headings of procedures and functions whose complete specifications appear in the <implement_part> of the module. It is exactly the items in the <export_part> which become accessible to any other code which subsequently imports the module.

```
<export_part>
   -> 'EXPORT' <qualifier>*  ( <export_declaration> list ';' )

<qualifier>
   -> 'HIDDEN' | 'QUALIFIED' | 'READONLY'

<export_declaration>
   -> ( <constant_definition_part> | <type_definition_part> |
                    <variable_declaration_part> )*
      ( <routine_heading> List ';' )  ';'?
```

There can be multiple <export_part>s in a module. This allows for different qualifiers to be used for different exported entities. Multiple qualifiers can also be applied to an export declaration. For example, one may want to make a list of variables 'qualified' 'readonly'.

'Hidden' is a qualifier which only has meaning for types. Its use on other entities is ignored. A hidden type behaves as if only its name and size are known outside the module. When a hidden type is imported it can only be used in certain ways. Variables of the type can be declared. Other types can be declared equivalent to it. Variables of the same type (identical or equivalent) may be assigned to each other (if assignable). Variables of hidden types may be passed by value (if assignment compatible as defined above) or reference. It is not possible to do anything which would require knowing the structure of the type, such as field references, indexing, or assignment of expressions or constants. Any information about the internal structure of the type is not exported. Type coercion at any level except for structural compatibility is allowed.

'Readonly' only applies to variables. Its use on other entities is ignored. A readonly variable may not be altered by assignment, nor passed as a Var parameter to a procedure or function.

Names which are exported 'qualified' can only be used in the context of the name of the exporting module. This allows two modules to export names which have identical spelling to the same program or an importing program to have identically spelled names.

Whenever the name of something exported from the module is used it must be qualified either with an explicit qualification (modulename ! entityname) or by a WITH statement. The semantics of the With statment is expanded from Pascal to allow module names to appear on the with list, syntax below (NOT IMPLEMENTED). The same scoping rules apply to the modules names as to the HP Pascal record expression list. Module names and record names can both appear in the same With list. The syntax of identifiers is expanded from Pascal to allow any identifier to be qualified using the following syntax.

```
<with__list>
-> ( ( <module__name> '.') | <record__variable> ) LIST ','


<identifier>
 -> ( <module__name> '.' )? <entity__name>
```

The modulename must be an imported module and entityname must be a constant, type, variable, procedure or function name exported by that module.

NOTE: 'QUALIFIED' most likely will go away!

NOTE: MODCAL/3000 will not support <variable_definition_part> code generation in this release.

The functionality or capability of the module -- what it can do for an importer -- appears in the <implement_part>.

```
<implement_part>
   ->  'IMPLEMENT' <module_capability>

<module_capability>
   ->  <declaration_part>
           <procedure_and_function_declaration_part>
```

Note: <declaration_part> and <procedure_and_function-declaration_part> are defined as in HP Standard Pascal. If <declaration_part> is not empty then there must be a <procedure_and_function_declaration_part> also.

Examining the syntax of <declaration_part> and <procedure_and_function_declaration_part> reveals that the module capability may be empty.

Any constants, types and variables declared in the implement part will not be made known to importers of the modules; they are only useful inside the module; outside of the module they are concealed. To make them known outside the module, they must be declared in the <export_part>. Variables of the <implement_part> of a module have the same lifetime as global program variables, even though they are concealed. An identifier which is concealed in one module may be reused as a concealed identifier in another module, and the two are totally disjoint.

Any procedures or functions whose headings are exported by the module must subsequently be completely specified in its <implement_part>. In this respect the headings in the export part are like forward directives, and in fact the parameter list of such procedures need not be repeated in the <implement_part>. Procedures and functions which are not exported may be declared in the <implement_part>; they are known and useful only within the module.

In the following example, module "Symboltable" is declared to implement a generalized symbol table. This module defines the only operations which can be performed on a symbol table by exporting procedures "Add" and "Delete". The type of the "data" in the symbol table, and the maximum size of its "name" field, are provided by module "Lexic". Since Symboltable depends on Lexic, Lexic must be imported into Symboltable. Hence Lexic must be compiled first.

```
MODULE Lexic;
EXPORT
  CONST  idsize = 16;
  TYPE   alpha  = PACKED ARRAY [1..idsize] OF char;
         attr   = RECORD
                     level: integer;
                     name:  alpha
                   END;
IMPLEMENT
  { empty <implement_part> since Lexic exports no procedures or
    functions }
END;

MODULE Symboltable;
IMPORT Lexic;
EXPORT
  PROCEDURE Add (ident: alpha; attrib: attr);
  PROCEDURE Delete (ident: alpha);
  PROCEDURE Initsymboltable;
IMPLEMENT
  TYPE symptr = ^symbol;
       symbol = RECORD
                   link: symptr;
                   name: alpha;
                   data: attr
                 END;
  VAR symtab: symptr;

  PROCEDURE Initsymboltable;
  BEGIN symtab := NIL  END;

  PROCEDURE Add (ident: alpha; attrib: attr);
  BEGIN
    { body omitted for clarity }
  END;

  PROCEDURE Delete; (parameter list need not be repeated)
  BEGIN
    { body omitted for clarity }
  END;
END.
```

e. Libraries <u>NOT IMPLEMENTED</u>

The modules Lexic and Symboltable as presented above illustrate that modules can always be compiled without reference to a program. In fact, a module can directly refer to variables of a program only if those variables are passed as parameters to procedures exported from the module. This restriction is not so severe as it may sound, since a module can be set up to export global variables shared among the program and other modules.

Separately compiled modules are called "library modules". To use library modules a program imports them just as if they had appeared in the program block:

```
PROGRAM USER;
IMPORT Lexic, Symboltable;
VAR
  instring: alpha;
  characterize: attr;

  PROCEDURE Scanner (VAR incoming: alpha; VAR description: attr);
  BEGIN
    { code to scan a symbol and set up its attributes }
  END;
BEGIN
END.
```

When an import declaration is seen, a module must be found matching each name in the import list. If a module of the required name appears in the compilation unit before the import declaration, the reference is to that module. Otherwise, external libraries must be searched.

The compiler option $SEARCH 'string'$ names the order in which externals libraries are searched. The parameter is a literal string describing the external libraries in an implementation-dependent fashion; usually the string will be a list of file names. This option may appear anywhere in a compilation unit, and overrides any previous SEARCH option. An implementation may have a default library which will be searched upon failure of all other libraries listed in 'string', such as the system library.

Note: How SEARCH works in different implementations is not yet defined.

3. Try Recover Statement

This statement defines error recovery code to be executed if an execution error is detected within a controlled statement.

a. Try Recover Structure

```
<structured_statement>
        ->  <compound_statement>
        ->  <conditional_statement>
        ->  <repetitive_statement>
        ->  <with_statement>
        ->  <try_recover_statement>

<try_recover_statement>
   ->  'TRY' <statement_list> 'RECOVER' <unlabelled_statement>
```

b. Try Recover Semantics

On detecting an error in the execution of <statement_list> the following sequence of events occur:

1) The Escapecode for the error is set,

2) the runtime environment is cut back to the enviroment in which the Try statement occurs,

3) Execution is transferred to the <unlabelled_statement> after the Recover.

If no error is detected in the execution of <statement_list> the <unlabelled_statement> is skipped, and execution continues at the first statement following the Try statement.

This construct divides the responsibilities of error recovery; error recovery which requires the enviroment in which the error occurs should use a trap handler provided by the system; for error reporting, the Try construct should be used.

If an error occurs in the execution of the <unlabelled_statement> , then execution is transferred to the enclosing Try statement. If there is no enclosing Try statement, then the program aborts.

NOTE: Try Recover statements are not permited in code that will execute on the ICS, for MODCAL/VCF.

c. *Escape*(i)

Calling this predefined procedure indicates that a software error has been detected. Execution is passed to the <unlabelled_statment> part of the first enclosing Try statement in the dynamic environment. The parameter i is evaluated before control is passed and its value is made available to the Escapecode function defined in (3.d). Errors detected by hardware will cause an *Escape* to be executed with a standard value. If *Escape* is called with no surrounding try statement, then the action taken is to abort the program.

WARNING: *Escape* will be 2-4 orders of magnitude more expensive then just entering and leaving a TRY statement normally,

d. *Escapecode*: integer

Calling this function returns the execution error number set by the most recent *Escape* call. If *Escape* has never been called the value zero (0) will be returned. If the <unlabelled_statement> passes control to the statement following the <try_recover_statement> in normal sequential manner, then the escape code will be reset to zero (0).

e. Error Code Assignment

The error code numbers are assigned using the following convention: The first 16 bits signifies which subsytem reported the error. The negative range has been reserved for HP products where the positive range is for user. The second 16 bits indicates which error occurred. The constants for HP products are defined in two include files, one for Vision and one for 3000.

f. Example

The following program illustrates a use of the try statement for error recovery in a hypothetical system. This system defines a set of run time support procedures (only one is shown) and a user program which may cause errors. If an error is detected in the user program by the hardware or in the support procedure by software, an *Escape* will be generated which causes execution to continue in the recover section of the main program. This section prints out an error message.

```
PROGRAM xx;
CONST
   $Include 'ModCode.Pub.Sys'$

  PROCEDURE support;
  BEGIN
  IF error THEN Escape( ... )
  END;

  PROCEDURE userprogram;
  BEGIN
  support
  END;

BEGIN
TRY userprogram
RECOVER
  CASE Escapecode OF
    MinUser..MaxUser: Writeln('Software detected errors');
    Lang1Rng        : Writeln('Value range error');
    SysStkOvf       : Writeln('Stack overflow');
    SysIntOvf       : Writeln('Integer overflow');
    SysIntDiv       : Writeln('Integer divide by zero');
    SysFltOvf       : Writeln('Real overflow');
    SysFltUnf       : Writeln('Real underflow');
    SysFltDiv       : Writeln('Real divide by zero');
    Lang1Nil        : Writeln('Nil pointer reference');
    Lang1Case       : Writeln('Case expression bounds error');
    Lang1StrOvf     : Writeln('String overflow');
    FileErr         : Writeln('File I/O error');
  OTHERWISE
    Writeln('unknown error')
  END;
END.
```

The Escapecodes used in the example are not the actual constants for the HP 3000 or VCF.

4. Type Coercion

Type coercion allows a programmer to circumvent the strong type checking of Pascal. The type coercion mechanism is a compile time operation (except in some cases of discriminated schema acccess) for applying a different interpetation to the data value part of a data item (Appendix B).

a. Syntax/Semantics

Type coercion has two (2) syntactic/semantic forms:

Value Type Coercion (no restrictions)

```
<factor>
   ->  <target_type_id>  '(' <expression> ')'
```

This form of type coercion allows the value of an expression to be coerced to the target type. The evaluation of the expression may create a copy of the value to be coerced, thus any machine dependent addressing problems can be avoided.

Variable Type Coercion (machine dependent addressing restrictions)

```
<variable>
   ->  <target_type_id>  '(' <variable> ')'
```

This form of type coercion allows the source variable to be coerced to the target type. Variable type coercion is permitted only for actual parameters. The source variable may not be addressable as the target type on a given machine (e.g. a byte as a word on the 3000). When this occurs the type coercion is not allowed and an error is issued. Three cases that are very likely to result in this condition are:

- Coercing any component of any CRUNCHED structure.

- Coercing any component of any PACKED structure whose component type *BitSizeof* is less than the *BitSizeof* of a SU.

- Coercing any component of any array whose component type *BitSizeof* is less than the *BitSizeof* of a SU.

b. Compatibility Levels

The use of type coercion can lead to the development of untransportable system software. Additionally one form of type coercion can result in the use or modification of storage beyond that occupied by the expression or variable coerced. These problems can be reduced by defining several levels of type coercion that progress from safe and transportable to very dangerous and untransportable. The four (4) levels of type coercion defined in MODCAL based on the compatibilities in Appendix C are:

STRUCTURAL - This level permits any data item to be coerced to any structurally compatible data type. This level simply permits renaming components. It is safe and fully transportable.

REPRESENTATION - This level permits any data item to be coerced to any representation size compatible data type. This level requires identical *BitSizeof* values. It is unsafe only if reference types are involved (Appendix B). It is transportable because all untransportable uses are flagged as errors.

STORAGE - This level permits any data item to be coerced to any storage size compatible data type. This level requires less or equal *Sizeof* values. It is unsafe and many untransportable uses will NOT be flagged as errors.

NONCOMPATIBLE - This level permits any data item to be coerced to any data type, providing that syntactic/semantic form of type coercion is defined on the target machine. This level is very dangerous and no errors can be flagged.

The safest level sufficient to permit the required type coercions is selected with the $Type_Coercion 'string'$ compiler option (Appendix A).

c. Example

```
MODULE MPE_File_System;

IMPORT
   MPE_IO_System,
   MPE_Kernal;

EXPORT
   TYPE
      Word = ShortInt;
      WordArray = ARRAY [1..1] OF Word;
      TargetLen = ShortInt;
      FileNumbers = ShortInt;

   FUNCTION FRead  (
          FileNum : FileNumber;
      VAR Target  : WordArray;
          TCount  : TargetLen
      ): TargetLen;
```

```
IMPLEMENT
   TYPE
      ArrayLen = 1..32767;
      FileSet  = SET OF FileNumbers;
   CONST
      ValidFiles = FileSet [1..255];

   FUNCTION BoundsCheck  (
           Origin    : OffsetSU;
           Length    : ArrayLen;
           ParmList  : OffsetSU
      ): BOOLEAN
      OPTION INLINE;

      BEGIN { BoundsCheck }
     $Type_Coercion 'REPRESENTATION'$
      IF (Word(GetDL) <= Word(Origin))
            AND                             { Value Type Coercion }
         (Word(ParmList) > Word(Origin)+Pred(Length))
     $Type_Coercion 'NONE'$
      THEN
         BoundsCheck := TRUE
      ELSE
         BoundsCheck := FALSE;
      END; { BoundsCheck }

    FUNCTION FRead  (
            FileNum  : FileNumber;
       VAR Target   : WordArray;
            TCount   : TargetLen
       ): TargetLen;
    VAR
       BuffLen : ArrayLen;
       ByteLen : TargetLen;

       PROCEDURE FRead_1  (
          READONLY BuffLen : ArrayLen
          )
          OPTION INLINE;
       TYPE
          WordSchema (Len : ArrayLen) = ARRAY [1..Len] OF Word;
          WordArray = WordSchema (BuffLen);

          PROCEDURE FRead_2  (
             VAR Buffer : WordSchema
             )
             OPTION INLINE;

             BEGIN { FRead_2 }
             { body omitted }
             END; { FRead_2 }
          BEGIN { FRead_1 }
         $Type_Coercion 'NONCOMPATIBLE'$
          FRead_2 (WordArray (Target)); { Varaible Type Coercion }
         $Type_Coercion 'NONE'$
          END; { FRead_1 }


       BEGIN { FRead }
       IF FileNum IN ValidFiles THEN
          BEGIN { have a file number }
          IF TCount < 0 THEN
             BEGIN { byte target count }
             ByteLen := -TCount;
             BuffLen := ByteLen DIV 2 + ByteLen MOD 2;
             END { byte target count }
          ELSE
             BEGIN { word target count }
             ByteLen := 0;
             BuffLen := TCount;
             END; { word target count }

         $Type_Coercion 'REPRESENTATION'$
          IF BoundsCheck (OffsetSU(Offsetpart (Addr(Target)))
             , BuffLen, AddtoOffset (GetQ , -6))
         $Type_Coercion 'NONE'$
          THEN
             BEGIN { target in stack }
             FRead_1 (BuffLen);
             END { target in stack }
          ELSE
             FRead := 0;
          END { have a file number }
       ELSE
          FRead := 0;
       END; { FRead }

  END.
```

5. Type Transfer function *ReOrd*

Type transfer allows a programmer to convert the representation of one type into the representation of another type. Type transfer differs from type coercion when the representation of one value is changed to another representation. For example, converting an integer expression to an enumerated type. The predefined function *ReOrd* performs the type transfer.

```
        Function  ReOrd (ordtype, expression): ordtype
```

Example:

```
PROGRAM ReOrd_Example;
TYPE
  numbers = (zero,one,two,three,four,five,six,seven,eight,
             nine);
  digit_range = 0 .. 9;

VAR
  mnemonic: numbers;
  binary_value: digit_range;

   BEGIN
   binary_value := 7;
   mnemonic := ReOrd (numbers,binary_value);
   (*  mnemonic is now equal to enumerated value seven *)
   mnemonic := ReOrd (numbers,binary_value + 1);
   (*  mnemonic is now equal to enumerated value eight *)
   Assert (mnemonic = ReOrd (numbers,Ord (mnemonic), 0);
   END.
```

6. Procedural/Functional Variables

a. Procedural Variables

1. Type Procedure

The Type definition syntax of Pascal is extended to allow the specification of procedural datatypes. This allows the creation and manipulation of procedural variables, and augments the parameter procedure mechanism already present in Pascal. A procedural type may be specified by using the keyword procedure followed by an optional parameter list as a type specifier. Variables of procedural types may be assigned procedures which have congruent parameter lists as defined in HP Pascal. Any procedure assigned must have the same or wider scope than the variable it is assigned to. **To assign a procedure to a procedural variable, the procedure name is used as a parameter to the *Addr* function.** See example below. A procedural variable can be assigned NIL. This means no procedure has been assigned.

2. *Call*

The predefined procedure "*Call*(Pvar, parm, parm, ...)" causes the indicated procedure to be called with the indicated parameters. Pvar is the identifier of a variable of type procedure. If during the execution of the procedure of a procedural variable, it accesses any non-local variables, then the variables accessed are the variables which were accessible at the time the procedure was assigned to the procedural variable. It is an error if Pvar has the value NIL or is undefined. It may not be possible to detect an undefined procedure variable.

3. Procedural Type Operations Model

```
 *------*
 |      | :=
 |      V
+-------------+                        ***************
| Procedural  |                        *   Actual    *
|  Variable   |                        *  Procedure  *
+-------------+                        ***************
 A      |                                     |
 |      |        /---------------             |
 | :=   | (parm) / Addr(Proc_Name)            | (parm)
 |      |        \ Addr(Proc_Parm)            |
 |      V         \---------------            V
+-------------+                        +-------------+
| Procedural  |                        |  Procedure  |
|  Parameter  |                        |  Parameter  |
+-------------+                        +-------------+
 A      |                               A      |
 |      | (parm)                        |      | (parm)
 *------*                               *------*
```

```
Proc_Parm ( ... )        }
Call(Prcdrl_Var,...)     } = { Proc_Name ( ... )
Call(Prcdrl_Parm,...)    }
```

4. Example

This program illustrates a use of a procedural type. First, type "p" is declared to be a procedural type with two real parameters. Subsequently, two procedures and one variable are declared which are compatible with this type. In the main program, the value of "pv" is set based upon some calculation and then the procedure "*Call*" is used to call the procedure referred to by "pv". Note that the identifiers in the parameter lists need not be identical.

```
PROGRAM ProcVar(output);

TYPE p = PROCEDURE (r1,r2: real);

VAR pv : p;
    test : boolean;

  PROCEDURE proc1(num,denom: real);

  BEGIN
  Writeln(num/denom);
  END;

  PROCEDURE proc2(m1,m2: real);
  BEGIN
```

```
    Writeln(m1*m2);
    END;

  BEGIN
  ...
  IF test THEN
    pv := ADDR(proc1)
  ELSE
    pv := ADDR(proc2);
  Call(pv,3.1,4.34);
  END.
```

b. Functional Variables

1. Type Function

The Type definition syntax of Pascal is extended to allow the specification of functional datatypes. This allows the creation and manipulation of function variables, and augments the parameter function mechanism already present in Pascal. A function type may be specified by using the keyword Function followed by an optional parameter list and a required colon and result type identifier as a type specifier. Variables of function types may be assigned functions which have congruent parameter lists as defined in HP Pascal and identical result types. Any function assigned must have the same or wider scope that the variable it is assigned to. To assign a function to a function variable, the function name is used as a parameter to the *Addr* function. See example below. The reserved word NIL may be assigned to a function variable. This means that no function is assigned to it.

2. *FCall*

The predefined function "*FCall*(Fvar, parm, parm, ...)" causes the indicated function to be called with the indicated parameters. Fvar is the identifier of a variable of type function. The type of the result of FCall is identical to the result type of Fvar. If during the execution of the function of a function variable, it accesses any non-local variables, then the variables accessed are the variables which were accessible at the time the function was assigned to the function variable. It is an error if Fvar is NIL or undefined.

3. Functional Type Operations Model

```
  *------*
  |      | :=
  |      V
+-------------+                                      ***************
| Functional  |                                      *   Actual    *
|  Variable   |                                      *  Function   *
+-------------+                                      ***************
  A      |                                                  |
  |      |             /----------------                    |
  | :=   | (parm)     / Addr(Func_Name)                     | (parm)
  |      |            \ Addr(Func_Parm)                     |
  |      V             \----------------                    V
+-------------+                                      +-------------+
| Functional  |                                      |  Function   |
|  Parameter  |                                      |  Parameter  |
+-------------+                                      +-------------+
  A      |                                             A      |
  |      | (parm)                                      |      | (parm)
  *------*                                             *------*
```

```
Func_Parm ( ... )        }
FCall(Fnctnl_Var,...)    } = { Func_Name ( ... )
FCall(Fnctnl_Parm,...)   }
```

4. Example

This program illustrates a use of a function type. First, type "f" is declared to be a function type with two integer parameters. Subsequently, two functions and one variable are declared which are compatible with this type. In the main program, the value of "fv" is set based upon some calculation and then the function "*FCall*" is used to call the function referred to by "fv". Note that the identifiers in the parameter lists need not be identical.

```
PROGRAM FuncVar;

TYPE f = FUNCTION(INT1,INT2: integer):integer;

VAR fv : f;
    test : boolean;
    Answer:integer;

  FUNCTION func1(num,denom: integer):integer;
    BEGIN
    func1:=num DIV denom;
    END;

  FUNCTION func2(m1,m2: integer):integer;
    BEGIN
```

```
        func2:=m1*m2;
        END;

    BEGIN
    ...
    IF test THEN
      fv := Addr(func1)
    ELSE
      fv := Addr(func2);
    Answer:=FCall(fv,3,4);
    END.
```

7. Special Predefined Functions

a. *Sizeof*

The *Sizeof* function returns the number of bytes needed to represent the data value part of a data item of the given type or the actual allocated size of a variable.

Sizeof(t) Gives the number of bytes required to represent a variable of type t.

Sizeof(v) Gives the number of bytes required to represent variable v or the actual number of bytes allocated to represent variable v.

Sizeof(p) Gives the number of bytes required to represent ANYVAR parameter p or the actual number of bytes allocated to represent the actual parameter passed to ANYVAR parameter p.

Sizeof(t1, t2, ... ,tn) Gives the number of bytes required to represent a variable of type t1 with tag field values t2, ..., tn.

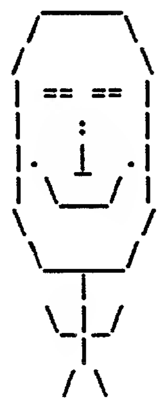
Sizeof(s1, s2, ... ,sn) Gives the number of bytes required to represent a variable of the discriminated schema type selected from the schema array s1 with discriminant values s2,...,sn (Section D).

b. *Addr*

The *Addr* function returns the reference part of the data item given to it as a parameter.

Addr(V) returns a pointer to the variable, V. It returns a pointer of the same type as NIL; that is, it is assignment compatible to any other pointer. Implementations may restrict its result to a pointer of type ^type of V. It is implemented this way in MODCAL/3000 and MODCAL/VCF. The pointer returned can be used as a parameter to the *ObjectPart* and *OffsetPart* pointer arithmetic functions.

B. HP MODCAL Standardized and Adaptable



C. HP MODCAL Standardized and NonAdaptable

```
      ______
     /      \
    /        \
   | ** **    |
   |    :     |
   |   _|_    |
   |   ____   |
   \  /    \  /
    \______/
        |
        |-
      _ |
     /  | \
       / \
```

D. Experimental

1. Read Only Parameters

The current parameter passing mechanisms of HP Standard Pascal supports only "call-by-value" and "call-by-reference", denoted by VAR. The first protects the actual parameter from modification, but requires making a copy. The second does not make a copy, but provides no protection for the actual parameter.

MODCAL provides a third mechanism "call-by-value-reference", denoted READONLY. This mechanism protects the actual parameter from modification, and makes a copy if and only if the actual parameter is an expression or a constant. No modification of a variable (actual parameter) is permitted between the call to and return from a routine with a READONLY (formal) parameter.

a. Read Only Parameter Definition

The following shows the syntax for defining read only formal parameters.

```
<formal__parm__section>
   ->  <value__parm__spec>
   ->  <variable__parm__spec>
   ->  <read__only__parm__spec>
   ->  <procedure__parm__spec>
   ->  <function__parm__spec>

<variable__parm__spec>
   -> 'VAR' <identifier__list> ':' <type__id>

<read__only__parm__spec>
   -> 'READONLY' <identifier__list> ':' <type__id>
```

b. Read Only Parameter Example

```
TYPE
  ArrType = ARRAY [1..10] OF INTEGER;
CONST
  ArrConst = ArrType [10 OF 0];
VAR
  ArrVar : ArrType;

  FUNCTION ArrFunc : ArrType;
    EXTERNAL;

  PROCEDURE Parm_Mechanisms (
          Value_Parm   : ArrType;
    VAR     Ref_Parm    : ArrType;
    READONLY Val_Ref_Parm : ArrType
    );
    EXTERNAL;

  BEGIN
  Parm_Mechanisms (
    ArrConst,  {a copy is made}
    ArrConst,  {an ERROR}
    ArrConst   {a copy is made & address is passed}
    );

  Parm_Mechanisms (
    ArrVar,  {a copy is made}
    ArrVar,  {address is passed}
    ArrVar   {address is passed}
    );

  Parm_Mechanisms (
    ArrFunc,  {a copy is made}
    ArrFunc,  {an ERROR}
    ArrFunc   {a copy is made & address is passed}
    );
  END;
```

2. Any Variable Parameters <u>NOT IMPLEMENTED</u>

The current "call-by-reference", denoted VAR, parameter passing mechanism of HP Standard Pascal requires any actual parameter be of the SAME TYPE as the formal parameter.

MODCAL provides another mechanism "call-by-any-reference", denoted ANYVAR. This mechanism removes the SAME TYPE requirement while retaining all other requirements of the "call-by-reference" mechanism (e.g. not passing components of PACKED or CRUNCHED structures).

This feature permits some very dangerous programming practices. Some MODCAL implementation may provide for some safety by creating a "phantom" parameter who value is the *SUSizeof* its companion ANYVAR actual parameter. This "phantom" can then be used by checking code and the *Sizeof, SUSizeof, and BitSizeof* functions (see Section E and Appendix A).

a. Any Variable Parameter Definition

The following shows the syntax for defining any variable formal parameters.

```
<formal__parm__section>
   -> <value__parm__spec>
   -> <variable__parm__spec>
   -> <anyvar__parm__spec>
   -> <procedure__parm__spec>
   -> <function__parm__spec>

<variable__parm__spec>
   -> 'VAR' <identifier__list> ':' <type__id>

<anyvar__parm__spec>
   -> 'ANYVAR' <identifier__list> ':' <type__id>
```

STRING must not be specified as the <type__id> when ANYVAR is specified.

b. Any Variable Parameter Example

```
TYPE
  ArrType1 = ARRAY [1..10] OF INTEGER;
  ArrType2 = ARRAY [1..20] OF INTEGER;
  ArrType3 = ARRAY [1..11] OF REAL;
VAR
  ArrVar1 : ArrType1;
  ArrVar2 : ArrType2;
  ArrVar3 : ArrType3;


  PROCEDURE Parm__Mechanisms (
    VAR    Ref__Parm1    : ArrType1;
    VAR    Ref__Parm2    : ArrType2;
    ANYVAR Any__Ref__Parm : ArrType2
    );
    EXTERNAL;

  BEGIN
  Parm__Mechanisms (
    ArrVar1, {ok - std. Pascal}
    ArrVar1, {an ERROR}
    ArrVar1  {ok / an ERROR for Any__Ref__Parm[11]}
    );

  Parm__Mechanisms (
    ArrVar2, {an ERROR}
    ArrVar2, {ok - std. Pascal}
    ArrVar2  {ok}
    );

  Parm__Mechanisms (
    ArrVar3, {an ERROR}
    ArrVar3, {an ERROR}
    ArrVar3  {ok / an ERROR for Any__Ref__Parm[12]}
    );
  END;
```

```
PROCEDURE Same__As__ANYVAR__chkd (
  VAR    Ref__Parm      : ArrType2;
       Ref__Parm__Len : ObjectSize;
  ANYVAR Any__Ref__Parm : ArrType2
  );
  EXTERNAL;

PROCEDURE Same__As__ANYVAR__unchkd (
  VAR    Ref__Parm      : ArrType2;
  ANYVAR Any__Ref__Parm : ArrType2
  )
  OPTION Uncheckable__ANYVAR;
  EXTERNAL;

BEGIN
Same__As__ANYVAR__chkd (
              $push, type__coercion 'noncompatible'$
  ArrType2(ArrVar1),
              $pop$
  SUSizeof (ArrVar1), {allocated SU's for Ref__Parm}
  ArrVar1
  );

Same__As__ANYVAR__unchkd (
              $push, type__coercion 'noncompatible'$
  ArrType2(ArrVar1),
              $pop$
                {allocated SU's for Ref__Parm UNKNOWN}
  ArrVar1
  );
END;
```

3. Schema Arrays <u>NOT IMPLEMENTED</u>

Types in Pascal are used for determining the storage requirements of variables and for checking the compatibility of assignments and procedure/function parameters. This double duty use of the type concept is too restrictive for some areas of system programming. To alleviate some of these restrictions, schema arrays have been added to MODCAL.

A schema can be thought of as a collection of types; each member of the collection is related to the other members in that they each have the same overall structure. The structure of each type is that of an array with the same component type and packing. However, each array type has a different index type.

MODCAL permits a formal parameter of a procedure or function to specify that it will accept any actual parameter whose type is a member of a specified schema. In this way the procedure or function can operate on a number of arrays with different index types, although only from the same schema.

Through the use of schemata, MODCAL also allows local arrays declared within a procedure or function to have their bounds specified at runtime. This permits the procedure or function to adapt its activation storage requirements in cases where local work areas are required.

a. Schema Array Definition

```
<type__definition__part>
    -> <type__def__section> ?

<type__def__section>
    -> 'TYPE'
         ( <type__definition> | <schema__definition> ) list ';'
```

This says that the <type__definition__part> of a block is composed of any number of type and schema definitions.

```
<new__type>
   -> <new__ordinal__type>
   -> <new__structured__type>
   -> <new__pointer__type>
   -> <discriminated__schema>
```

This specifies that a <new__type> may be created by any of the means existing in Pascal or by selecting one of the members of a schema. Selecting one of the members of a schema is called *discriminating* a schema, that is, specifying its actual index types.

A schema-definition shall introduce an identifier to denote a schema. A schema defines a collection of <new__type>s whose type-denoter is a discriminated schema.

```
<schema__definition>
    -> <identifier> <formal__discriminant__part>
            '=' <array__schema>

<formal__discriminant__part>
    -> '(' <discriminant__spec> list ';' ')'

<discriminant__spec>
    -> <identifier__list> ':' <ordinal__type__id>

<array__schema>
    -> ( 'PACKED' | 'CRUNCHED' )?
          'ARRAY' '['
            (<schema__index__type> list ',')
              ']' 'of' <component__type>

<schema__index__type>
    -> ( <constant> | <discriminant__id> )
          '..' ( <constant> | <discriminant__id> )

<discriminant__id>
    -> <identifier>
```

The occurrence of an identifier in a <schema__definition> of a <type__definition__part> shall constitute its defining point for the region that is a block. Each applied occurrence of that identifier shall denote the same schema. Except for applied occurrences of the identifier as the domain type of a pointer type, the schema shall not contain an applied occurrence of the schema definition.

NOTE: Currently, the <schema__index__type> must be an explicit subrange; therefore, it may not be the <type__id> of a subrange type.

The above definitions add the mechanism by which to define a schema. The leading identifier on the schema-definition becomes known. A schema may not have any references to itself except when used as the domain of a pointer. Thus, a schema has the same scope as a type declared at the same place.

b. Formal Discriminant Part

The <formal__discriminant__part> in a schema definition shall define the formal discriminants. The occurrence of an identifier in a <discriminant__spec> shall constitute its defining point as a <discriminant__id> for that region of the program that is the following <array__schema>.

For every <discriminant__id> in a <formal__discriminant__part>, there shall be at least one applied occurrence in the <array__schema>. The occurrence of a <discriminant__id> in a <schema__index__type> of an <array__schema> shall specify that there is one type denoter which is a member of the schema for each allowed value of the <discriminant__id> such that all other <schema__index__type> values in the schema are the same.

The <formal__discriminant__part> is used to associate identifiers with the schema so that the domain (members of the schema) can be determined. Every identifier used in the <formal__discriminant__part> must be used at least once in the following <array__schema>. In the following, *SmallVec* is a collection of ten type denoters with index types "0..1","0..2", ... , "0..10".

```
    type
      SmallInt = 1 .. 10;
      SmallVec (HighBound : SmallInt) =
        array [0 .. HighBound] of real;
```

c. Discriminated Schema

A <discriminated_schema> selects one of the members of a schema as a <new-type>. The <discriminant_value>s are bound to their corresponding <discriminant_specs> in the <formal_discriminant_part> of the schema. The number of discriminant values must be equal to the number of formal discriminants and each value must be assignment compatible with the type of the corresponding formal discriminant.

```
 <discriminated_schema>
     -> <schema_id> <actual_discriminant-part>

 <actual_discriminant_part>
     -> '(' <discriminant_value> list ',' ')'

 <discriminant_value>
     -> <constant_expression> |
               <read_only_variable>
```

A <discriminated_schema> appearing in the declaration part of a block must have only constant or read_only variable discriminant values. This restriction assures that evaluation of discriminant values in the declaration part of a block can have no side effects.

Any schema designated PACKED and denoting an array schema having its <schema_index_type> specifying as its smallest value a constant whose value is 1, and having as its component type a denotation of the char type, shall be a PAC schema. Any new type specifying a discriminated-schema which is a PAC schema shall be designated a PAC, and has all the special properties of a PAC defined in Pascal.

A discriminated schema is a type denoter selected from the collection of type denoters in the schema. The values given in the <actual_discriminant_part> are used (substituted) for the <formal_discriminant_part> in the <array_schema>. Thus the discriminated schema "SmallVec(7)" selects the member of the schema which is structurally compatible to (but not the same as) the array:

```
ARRAY [0 .. 7] OF REAL
```

An attempt to specify the schema as "SmallVec(11)" will result in an error because the value 11 is not assignment compatible with the type of Highbound.

It must be noted that although a <discriminated_schema> is equivalent in structure to an array type, it is never the same (in the sense of Pascal type compatibility). Moreover, two discriminated schemas that specify the same discriminant values are not the same.

Any <discriminated_schema> having the same number of components of structurally compatible types with the identical packing are said to be STRUCTURALLY compatible. Any unpacked arrays or unpacked discriminated-schemas having the same number of components of structurally compatible types are said to be structurally compatible (see Section A and Appendix C).

Any <discriminated_schema> with <read_only_variable>s in its <actual_discriminant_part> denotes a type that may be used only in a <variable_declaration_part> or a type coercion.

d. Schema Discriminants

```
<factor>
  -> <variable_access>
  -> <unsigned_constant>
  -> <function_designator>
  -> <set_constructor>
  -> '(' <expression> ')'
  -> 'NOT' <factor>
  -> <schema_discriminant>
  -> <read_only_variable>

<schema_discriminant>
  -> <variable_access> '.' <discriminant_id>

<read_only_variable>
  -> <variable_access>
```

The schema discriminants for a discriminated schema value are simply the values that were used to satisfy the formal discriminants in the declaration of the discriminated schema which is its type. The notation to access this value uses the name of the formal discriminant from the schema declaration. For example, consider the following schema, discriminated schema and discriminated schema value:

```
TYPE
  SmallInt = 1..10;
  Schema(Start,Length : SmallInt) = ARRAY[Start..Length] OF
                                    INTEGER;

  DiscSchema = Schema(1,5);
VAR
  X : SmallInt;
  DiscSchemaValue : DiscSchema
```

The actual discriminant values for *DiscSchemaValue* are 1 and 5, accessible by the notation *DiscSchemaValue.Start* and DiscSchemaValue.Length, respectively.

Because a factor can never appear as a target of an assignment, the discriminant may never be altered. The value of the discriminant could be thought of as a "read only variable" associated with the variable or parameter. The vector addition example above shows the use of <schema_discriminant>s.

e. Schema Array Parameters

Discriminant values can be constants or read only variables (variables whose value cannot be altered); thus discriminant values are constant over the entire scope of a discriminated schema. The following shows the syntax for defining read only variables and schema array formal parameters.

```
<formal_parm_section>
   -> <value_parm_spec>
   -> <variable_parm_spec>
   -> <read_only_parm_spec>
   -> <procedure_parm_spec>
   -> <function_parm_spec>

<variable_parm_spec>
   -> 'VAR' <identifier_list> ':'
              ( <type_id> | <schema_id> )

<read_only_parm_spec>
   -> 'READONLY' <identifier_list> ':'
              ( <type_id> | <schema_id> )
```

These productions deal with read only parameters and schema parameters. Read only parameters are discussed in the next section; this section deals exclusively with schema parameters.

A variable may be passed into a procedure or function whose type denoter is a member of a schema. When a schema identifier is specified, the parameter may be of any type which is a member of that schema.

If the formal parameters are specified in a <variable_parm_spec> in which there is a <schema_id>, the type possessed by the actual parameter shall be itself a parameter that was specified with the same <schema_id>; and the type possessed by the formal parameter shall be distinct from any other type.

This states that a formal parameter that was declared with a schema will only permit the actual parameter to be of a type which is a member of the same schema. A formal parameter which is a schema may in turn be passed on as a variable-parameter utilizing the same schema.

If the form of the parameter list includes an <identifier_list>, then all the actual parameters must be of the same type (only for variable parameters); this is true for schemas as well as other types.

The following example adds two vectors, element by element, and returns the result in the first parameter.

```
PROCEDURE AddVectors (VAR A,B,C : SmallVec);
VAR
  i : SmallInt;

  BEGIN
  Assert (A.HighBound = B.HighBound, 1);
  Assert (A.HighBound = C.HighBound, 2);
  FOR i := 0 TO B.HighBound DO
    A[i] := B[i] + C[i];
  END;
```

f. Read Only Parameters

The productions from the previous section state that read only variables are defined by <read_only_parm_spec>s (read only variables are also exported by modules). The occurrence of an identifier in the <identifier_list> of a <read_only_parm_spec> shall constitute its defining point as a read only variable for the region that is the block, if any, of which it is a formal parameter.

All parameters that are specified with the read only mechanism are identified as being read only variables; this permits them to be limited to being factors within the block. The actual parameter shall be an expression. A formal parameter declared with a schema as its type specification and occuring in a single <read_only_parm_spec> shall possess an array type which is distinct from any other type. The type possessed by the actual parameter shall be a discriminated schema designating the same <schema_id> as the formal_parameter or the actual_parameter shall be itself a parameter that was specified with the same schema identifier; or the actual parameter must be a PAC and the formal parameter must designate a PAC-schema; or the actual parameter must be of the same type as the formal-parameter.

For an actual parameter that denotes a read only variable access, there shall be no assigning reference during the activation of the block of the procedure or function to the actual parameter.

This introduces a parameter mechanism into MODCAL that requires that the actual_parameter may not be altered during the activation of the associated procedure or function. Any expression may be specified by the actual_parameter. Thus, the mechanism achieves not only protection of the actual-parameter but also permits literal strings or value type coercions to be specified.

g. Discriminated Schema Access

```
<discriminated_schema_access>
   -> <discriminated_schema_value>
         '[' <schema_access_list> ']'

<discriminated_schema_value>
   -> <variable_access>

<schema_access_list>
   -> ( <index_expression> ',' )* <schema_access_expr>

<schema_access_expr>
   -> <disriminant_value> 'FOR' <discriminant_length>

<index_expression>
   -> <expression>

<discriminant_length>
   -> <expression>
```

NOTE: Only <schema_access_List> -> <schema_access_expr> will be permitted in the next several releases. Also, the following restrictions are being considered: 1) Discriminated schema access only permitted with one-dimensional schemas and 2) <discriminant_length> must be a constant (e.g., DiscSchemaValue[x FOR 5]).

A <discriminated_schema_value> is any variable, parameter, or constant which has a discriminated schema as its type denoter. A <discriminated_schema_access> selects a contiguous sequence of the components of a <discriminated_schema_value>. This contiguous sequence has a type denoted by another member of the same uniquely discriminated schema as the <discriminated_schema_value>. A contiguous sequence of components of any array or <discriminated_schema> type is just a sequence of components that occupy consecutive storage units. That is, for array types stored in row major order, any subrange of the last index type of the array defines a contiguous sequence of its components. A subrange can also be specified for an index type other than the last one. In this case, no following index types may be specified; the entire ranges of the following index types are used in order to ensure contiguity. Consider the following schema, discriminated schema and discriminated schema value:

```
TYPE
  SmallInt = 1..10;
  Schema2D(L1,U1,L2,U2 : SmallInt) =      {A schema}
       Array[L1..U1,L2..U2] OF INTEGER;

  Square = Schema2D(1,5,1,5);          {A disc. schema}
VAR
  Square1 : Square;                {A disc. schema value}
```

The discriminated schema access *Square1[4, 3 FOR 3]* specifies the 3rd through 5th components of the 4th row of *Square1*. *Square1[2 FOR 2]* specifies the entire 2nd and 3rd rows of *Square1*. Clearly, if a schema access expression (*n* FOR *m*) were permitted for the second index position of this second example, the selected components would not be contiguous.

Also a discriminated schema access is a discriminated schema value; its type is a member of the same schema, but its index types are a subrange of the corresponding index types of the original discriminated schema value. For example, *Square1[4, 3 FOR 3]* is a discriminated schema value; its type is a discriminated schema with its first index type the subrange *4..4* and its second index type the subrange *3..5*. This discriminated schema is still a member of the schema *Schema2D*.

It was mentioned above that a discriminated schema access is only permitted with a uniquely discriminated schema. A uniquely discriminated schema is defined as any schema definition having two formal discriminants in the formal discriminant part for each schema index type in the array schema. This means that each of the upper and lower bounds of each index type has its own formal discriminant. So the schema *Schema2D* is uniquely discriminated, whereas the schema *SmallVec* from the preceding discussion of formal discriminants is not. We required earlier that a discriminated schema access itself be a member of the original schema. This is not guaranteed when the discriminated schema is not uniquely discriminated. To see this, consider again the schema *SmallVec*.

```
TYPE
  SmallInt = 0..10;
  SmallVec(HighBound : SmallInt) =  ARRAY[0..HighBound] OF REAL;
VAR
  Vec7 : SmallVec(7);
```

Now if we were allowed to use *Vec7* in a discriminated schema access, say *Vec7[3 FOR 4]*, then the type of the resulting discriminated schema access would not be a member of the schema *SmallVec*. Any member of *SmallVec* must have 0 as its lower bound; however, the type of *Vec7[3 FOR 4]* has 3 as its lower bound. Clearly, this is not a member of the schema *SmallVec*.

The following example illustrates the use of discriminated schema access, declaration of a uniquely discriminated schema, and schema array parameters.

```
PROGRAM Discrm_Schema_Access;
TYPE
   IndexType = 1..12;
  Schema (L1,U1,L2,U2 : IndexType) =
    ARRAY [L1..U1, L2..U2] OF ComponentType;
  DiscrSchema1 = Schema (1,10,1,12);
  DiscrSchema2 = Schema (3,3,2,8);
  DiscrSchema3 = Schema (2,4,1,12);
VAR
  DiscrSchema1Value : DiscrSchema1;
  DiscrSchema2Value : DiscrSchema2;
  DiscrSchema3Value : DiscrSchema3;

  PROCEDURE SchemaParm (VAR DiscrParm : Schema);
    BEGIN
    END;
```

```
BEGIN
SchemaParm (DiscrSchema1Value);
    (*DiscrParm.L1 = 1
      DiscrParm.U1 = 10
      DiscrParm.L2 = 1
      DiscrParm.U2 = 12*)

SchemaParm (DiscrSchema2Value);
SchemaParm (DiscrSchema1Value[3, 2 FOR 7]);
    (*DiscrParm.L1 = 3
      DiscrParm.U1 = 3
      DiscrParm.L2 = 2
      DiscrParm.U2 = 8*)

SchemaParm (DiscrSchema1Value[2 FOR 3]);
SchemaParm (DiscrSchema3Value);
    (*DiscrParm.L1 = 2
      DiscrParm.U1 = 4
      DiscrParm.L2 = 1
      DiscrParm.U2 = 12*)
END.
```

h. Example

```
PROGRAM ShowSchemaArrays;
CONST
  maxsize = 25;
  $include 'other.consts.example'$

TYPE
  Size = 1..Maxsize;
  Matrix(Row,Col : Size) = ARRAY [1..Row,1..Col] OF Real;

VAR
  A : Matrix(10,5);
  B : Matrix(5,10);

  PROCEDURE MatrixMultiply (
    VAR      A : Matrix;
    READONLY B : Matrix);
  VAR
    I,J,K : Size;
    T     : real;
    C     : Matrix(A.Row,B.Col);
```

```
BEGIN (* MatrixMultiply *)
IF (A.Row = B.Col)
     OR
   (A.Col = B.Row)
THEN
  BEGIN
  FOR I := 1 TO A.Row DO
    FOR J := 1 TO B.Col DO
      BEGIN
      T := 0.0;
      TRY
        FOR K := 1 TO A.Col DO
          T := T + A[I,J] * B[K,J];
      RECOVER
        CASE EscapeCode OF
          FltPtOverflow:
            T := MaxReal;
          FltPtUnderflow:
            T := MinReal;
          OTHERWISE
            Escape (EscapeCode);
          END;
      C[I,J] := T;
      END;
   FOR I := 1 TO A.Row DO
     FOR J := 1 TO A.Col DO
       A[I,J] := C[I,J]; (*return funky results*)
   END
 ELSE
   Escape (BadMatrix);
 END; (* MatrixMultiply *)

BEGIN
MatrixMultiply (A,B);
END.
```

4. Objects

a. Introduction

The HP 3000 and Vision Computer families are segmented machines. It is impossible to place Pascal variables in data segments/objects. So the type classes object and offset have been placed in MODCAL to correct this deficiency.

An object is a contiguous sequence of SU's numbered from zero to some maximum. An object may be dynamically created and destroyed, and its lifetime is independent of any Pascal scope. Upon creation, an object is given an initial size and a maximum size in SU's. An object may grow up to its maximum size and may shrink down to zero. Each object has a compile-time defined storage area in its first n SU's, where n is the size of this static area (which may be 0, e.g. RECORD END). The remaining space in the object may be used as the programmer chooses.

A pointer in MODCAL is composed of two parts: an object specifier and an offset. An object specifier is declared and used much in the same way that pointers are used. The symbol used for dereferencing an object specifier is '%', while the symbol used for dereferencing pointers is '^'. A dereferenced object specifier with an offset forms a pointer. An offset is useless in referencing data without a dereferenced object specifier.

Note: Heap pointers in MODCAL/3000 do not have the same underlying representation as an object, offset pair.

b. Syntax Additions

Objects require the following changes to the syntax charts for Pascal:

variable

```
----+---------------> variable id ------------------------->+
    |                                                       |
    +------------------> field id ------------------------->+
    |                                                       |
    +--> object var -> '%' -+-> '(' -> offset var -> ')' -->+
                            |                               |
                            V                               V
    +<----------------------+-------------------------------+
    |
    +<-----------------------------------------+
    |                                          |
    +--> '[' -+--> expression -+--> ']' ------->+
    |         ^                |               |
    |         |                V               |
    |         +<----- ',' -----+               |
    |                                          |
    +--> '.' ----> field id ------------------>+
    |                                          |
    +------------> '^' ----------------------->+
    |
    V
```

or in grammar notation:

```
<variable>
     ->  <object_var>  '%'  ( '(' <offset_var> ')' )?

<object_var>
     ->  <variable>

<offset_var>
     ->  <variable>
```

type

```
--+--> simple type ------------------------------------->+-->
  |                                                      ^
  +---------------+-+--> '^' -------------+-> type id ->+
  |               | |                     |              ^
  +<-- 'PACKED' --+ +--> '%' -------------+              |
  |               | |                     |              |
  +<- 'CRUNCHED' -+ +-> 'OFFSET' -> 'TO' -+              |
  |                                                      |
  +--> 'ARRAY' --> '[' -+--> simple type -+--> ']' -+    |
  |                     ^                 |         |    |
  |                     +-- ',' <---------+         |    |
  |                                                 |    |
  |  +<---------------------------------------------+    |
  |  |                                                   |
  |  +--> 'OF' --> type ------------------------------->+
  |                                                      ^
  +--> 'FILE' --> 'OF' --> type ----------------------->+
  |                                                      |
  +--> 'SET' --> 'OF' --> simple type ----------------->+
  |                                                      ^
  +--> 'RECORD' --> field list --> 'END' -------------->+
  |                                                      |
  +--> schema arrays ---------------------------------->+
```

or in grammar notation:

```
<type>
   ->  <simple_type>
   ->  ( 'PACKED' | 'CRUNCHED' )  <type>
   ->  ( '^' | '%' | 'OFFSET' 'TO' )   <type_id>
   ->  'ARRAY'  '['  ( <simple_type> list ',' )  ']'
                           'OF'  <type>
   ->  'FILE'  'OF'  <type>
   ->  'SET'  'OF'  <type>
   ->  'RECORD'  <field_list>  'END'
   ->  <schema_definition>
   ->  <discriminated_schema>

<statement>
    -> 'WITHOBJECT'  ( (<object_var> '%') list ',' )
                'DO'  <statement>
```

c. Object/Offset Model

```
        VAR obj1 : % type_name1;
            obj2 : % type_name2;
            off  : OFFSET TO type_name1;
            ptr1 : ^ type_name1;
            ptr2 : ^ type_name2;

        obj1 ====>++---------------++   ---
               |    ||  obj1%         ||    :
               |    ||                ||    :  Static Area
               |    ||{ type_name1  }||    :
              off   |+---------------+|    :
               |    |                 |    :
               |    |                 |    :
               |    |                 |    :
             ===>|+---------------+|    :
                    ||  obj1%(off)^  ||    :
                    ||                ||    :
                    ||{ type_name1  }|| GetObjectSize(obj1)
                    |+---------------+|    :
                    |                 |    :
        ptr1 ====>++---------------++    :
                    ||  ptr1^         ||    :
                    ||                ||    :
                    ||{ type_name1  }||    :
                    |+---------------+|    :
                    |                 |    :
                    +-----------------+   ---

        obj2 ====>++---------------++   ---
               |    ||  obj2%         ||    :
               |    ||{ type_name2  }||    :  Static Area
               |    |+---------------+|    :
              off   |                 |    :
               |    |                 |    :
               |    |                 |    :
               |    |                 |    :
             ===>|+---------------+|    :
                    ||  obj2%(off)^  ||    :
                    ||                || GetObjectSize(obj2)
                    ||{ type_name1  }||    :
                    |+---------------+|    :
                    |                 |    :
                    |                 |    :
        ptr2 ====>++---------------++    :
                    ||  ptr2^         ||    :
                    ||{ type_name2  }||    :
                    |+---------------+|    :
                    |                 |    :
                    +-----------------+   ---
```

d. Semantic Definition

Consider the following MODCAL block:

```
TYPE
   Foo = RECORD
      int1, int2: integer
      END;

   Obj       = %Foo;
   OffsetInt = OFFSET TO integer;

VAR
   systable : Obj;
   f        : Foo;
   off      : OffsetInt;
   errcode  : integer;

BEGIN
CreateObject(systable,100,100,errcode);
IF errcode = 0 THEN
   BEGIN
   systable%.int1 := 1;
   systable%.int2 := systable%.int1;
   f := systable%;

   MakeOffset(off, SUSizeof(Foo));
   WITHOBJECT systable% DO
      off^ := 7;
   END;
END;
```

This declares systable as an object variable of type 'Obj' with a static area containing two integers. The notation systable% denotes systable's static area and is a variable of type Foo. The notation systable%(off) denotes a pointer of type ^Integer. The first statement of the program stores a '1' into the first integer of the static area, and the second statement copies the value of the first integer into the value of the second integer in the static area. The third statement copies the entire static area into f. The final two statements have the effect of placing a seven following systable's static area. See the section on pointer arithmetic for the definition of *MakeOffset*.

As an abbreviated notation, a dereferenced object variable, systable%, can be used as a variable in a Withobject list. In this use, one can denote an offset in systable or part of systable's static area without having to qualify the offset or static area field name with systable for the scope of the Withobject statement.

Note: On the 3000, only one object variable can be used in a Withobject list.

The assignment and compatibility rules for objects and offsets are the same as for pointers in Pascal.

e. Object Manipulation

The following predefined procedures which operate on objects have been defined.

```
PROCEDURE Createobject (
   VAR object  : objectform;
       size    : ObjectSize;
       maxsize : ObjectSize;
   VAR status  : errorcode);
```

The var parameter object contains the object number of the object created. The parameter size is the initial allocation of space given to the object, and the parameter maxsize is the maximum number of storage units that the object may grow to. The var parameter status is an integer which indicates the success of the operation (zero (0) = success).

```
PROCEDURE AlterObjectsize (
       object  : objectform;
       newsize : ObjectSize;
   VAR status  : integer);
```

This predefined procedure alters the size of an object. The var parameter object is the object whose size is to altered, the value parameter newsize is the new size of the object, and the var parameter status is an errocode which indicates the success or failure of the operation (zero (0) = success). If the newsize is greater than the maxsize given in the call to Createobject, then the results are system dependent.

```
PROCEDURE DestroyObject (
   VAR object : objectform;
   VAR status : integer);
```

This predefined procedure returns an object to the operating system. The var parameter object is the object to be returned, and the var parameter status is an integer which indicates the success or failure of the operation (zero (0) = success).

Because target machines may provide more powerful operations on objects through operating system intrinsics or a user may want to gain access to an object maintained by the operating system or some other user, the following operation is provided.

```
PROCEDURE AttachObject (
   VAR object : objectform;
       id     : objectid;
   VAR status : integer);
```

or

```
PROCEDURE AttachObject (
   VAR object : objectform;
       id     : objectid );
```

This predefined procedure places the value parameter id into the var parameter object, and returns the success or failure of the operation in the var parameter status (zero (0) = success). It is assumed that id is the object number of an already existing operating system object.

```
FUNCTION Stack
   : objectform;
```

This predefined function returns an object that corresponds to the active process stack. The type of the object returned is structurally compatible with any object having a fixed part with *Sizeof* equal to zero.

```
FUNCTION CurrentObject
   : objectform;
```

This predefined function returns an object that corresponds to the object refence of the most resently executed WITHOBJECT clause. The type of the object returned is structurally compatible with any object having a fixed part with *Sizeof* equal to zero. MODCAL/3000 will insure that DB points to this object when any procedure or function with a directive of EXTERNAL SPL is called.

```
FUNCTION GetObjectId (
               Object: objectform
                     ): objectId;
```

This predefined function returns the "object id" of the object denoted by the value parameter object (basically the inverse of *AttachObject*).

```
FUNCTION GetObjectSize (
               Object: objectform
                     ): ObjectSize;
```

This predefined function returns the number of SU's currently allocated to the object denoted by the value parameter object. If *Stack* is passed to *GetObjectSize* the size of the currently accessable area is returned (HP3000 : S-DL, VISION : S-SB).

f. Pointer Arithmetic

The following predefined functions are provided to permit pointer arithmetic. The types ObjectSize and ObjectDelta are defined in the portability foundation section.

```
PROCEDURE MakeOffset (
   VAR offsetvar : offsetform;
       value     : ObjectDelta);
```

The var parameter offsetvar is made to point to the variable beginning at the storage unit position given in the value parameter value.

```
FUNCTION AddToOffset (
       offsetvar : offsetform;
       delta     : ObjectDelta
   ): offsetform;
```

This predefined function adds the value parameter delta to the value parameter offsetvar and returns this value. The type of offset returned is identical to the type of the offset parameter.

```
PROCEDURE MakePointer (
   VAR ptr    : pointerform;
       object : objectform;
       offset : offsetform);
```

This procedure constructs a pointer from the value parameters object and offset, and returns the constructed pointer in the var parameter ptr. The reference type of the offset type and the pointer type must be identical. MODCAL/3000 requires that the object parameter be equal to the object returned by *Stack*.

```
PROCEDURE BuildPointer (
   VAR ptr    : pointerform;
       object : ObjectId;
       offset : ObjectDelta);
```

This procedure constructs a pointer from the value parameters object and offset, and returns the constructed pointer in the var parameter ptr. MODCAL/3000 requires that the object parameter be equal 0. This procedure is intended to replace a sequence of: *AttachObject*, *MakeOffset*, *MakePointer*.

```
FUNCTION AddToPointer (
       ptr   : pointerform;
       delta : ObjectDelta
   ): pointerform;
```

This predefined function returns a pointer value which points delta storage units away from where the value parameter ptr originally pointed. The type of the pointer returned is identical to the type of the pointer parameter.

```
  FUNCTION OffsetPart (
         ptr : pointerform
     ): offsetform;
```

This predefined function returns the offset part of the pointer parameter. The offset has the same reference type as the pointer.

```
  FUNCTION ObjectPart (
         ptr : pointerform
     ): objectform;
```

This predefined function returns the object part of the pointer parameter. The object has the same reference type as the pointer.

The meta-types "pointerform", "objectform", and "offsetform" are used above to indicate that these predefined procedures and functions will accept and return any form of pointer type, object type, or offset type.

5. Crunch Packing

The word 'crunched' has been added to the reserved word list to indicate that the components of a structured type are allocated contiguously, first to last in a bit-aligned fashion. Components may cross any arbitrary machine storage unit boundry. The word 'crunched' may be substituted for the word 'packed'.

Example

```
TYPE t = CRUNCHED RECORD
  a : 0 .. 15;
  b : -32767 .. 32767;
  c : #0 .. #255; {currect Char subrange}
  END;
```

```
MSB                           LSB
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |a a a a|b b b b b b b b b b b|
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |b b b b|c c c c c c c c|XXX|
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Layout of type crunched type t (assuming 16 bit SUs)
(note: the value of XXX is undefined)

The number of bits used to represent each component of a crunched structured type is the minimum needed to represent the values associated with that component. The calculation for the minimum number of bits is:

```
Integer Based Types    (lo .. hi)
   if (lo <> 0) or (hi <> 0)
    then ceil(log2(max(abs(lo),succ(abs(hi))))) + ord(lo < 0)
    else 1

Character & Enumerated Based Types    (lo .. hi)
   if ord(hi) > 0
    then ceil(log2( succ(ord(hi)) ))
    else 1

Set Types    (SET OF lo .. hi)
   succ(abs(ord(hi) - ord(lo)))

Record, Array & Nonimbedded Descriptor Types
   The sum of the minimum number of bits to represent all
   of the components (consider largest variant only).
```

For example, it would take only four bits to represent a component with the subrange type 0 .. 15 or the subrange type 14 .. 15, but it will take five bits to represent a component with the subrange type -1 .. 14.

The restrictions which apply to packed types also apply to crunched types. This means that record comparisons are not allowed. In addition, it is not legal to pass by reference a component of a crunched structured type to any procedure or function.

The primary purpose of crunch packing is to provide an implementation invariant specification of the data item type to data item representation mapping. The implementation invariant requirement necessitates the restrictions listed below.

No file types or any structured type containing any file, real, string, or reference types may be crunch packed.

Any structured type contained in a crunched structured type must be crunch packed also.

No structured type containing an Integer or Char may be crunch packed. The range of these predefined types may be changed in the future. The user can always replace the types Integer and Char with a subrange type.

NOTE: Crunched structures are temporarily required to be less than or equal to 2048 words (16 k-bits) in length.

6. *Move_R_to_L* & *Move_L_to_R* Predefined Procedures

The Move predefined procedures provide generalized array structure copying mechanisms in MODCAL. The syntax of the Move procedures is identical to the syntax of the *StrMove* predefined procedure. The semantics of the Move procedures are presented below in a Meta-MODCAL form. The lower case identifiers denote the meta-parameters of the semantics. The actual parameters to a Move are (structurally) type coerced to match the formal parameters by the compiler.

```
TYPE
   COUNT_RANGE = 1 .. MAXINT;

   SOURCE_TYPE (MIN, MAX : INTEGER) =
      source_packing ARRAY [MIN .. MAX] OF component_type;

   TARGET_TYPE (MIN, MAX : INTEGER) =
      target_packing ARRAY [MIN .. MAX] OF component_type;

PROCEDURE move_r_to_l_move_l_to_r  (
            MOVE_CNT    : INTEGER;
   READONLY SOURCE      : SOURCE_TYPE;
            SOURCE_BIAS : INTEGER;
   VAR      TARGET      : TARGET_TYPE;
            TARGET_BIAS : INTEGER
   ) OPTION INLINE;
VAR
   LOOP_CNT,
   LOOP_LIMIT : COUNT_RANGE;
```

```
      BEGIN { MOVE Semantics }
      IF MOVE_CNT > 0 THEN
         BEGIN { Move Some Components }
         LOOP_LIMIT := Pred (MOVE_CNT);

     Assert (SOURCE_BIAS              >= SOURCE.MIN, 0, move_error);
     Assert (SOURCE_BIAS + LOOP_LIMIT <= SOURCE.MAX, 0, move_error);
     Assert (TARGET_BIAS              >= TARGET.MIN, 0, move_error);
     Assert (TARGET_BIAS + LOOP_LIMIT <= TARGET.MAX, 0, move_error);

         CASE move_r_to_l_move_l_to_r OF
          MOVE_R_TO_L:
            FOR LOOP_CNT := LOOPLIMIT DOWNTO 0 DO
               TARGET[LOOP_CNT + TARGET_BIAS] :=
                  SOURCE[LOOP_CNT + SOURCE_BIAS];

          MOVE_L_TO_R:
            FOR LOOP_CNT := 0 TO LOOP_LIMIT DO
               TARGET[LOOP_CNT + TARGET_BIAS] :=
                  SOURCE[LOOP_CNT + SOURCE_BIAS];
          END; { CASE }
         END; { Move Some Components }
      END; { MOVE Semantics }
```

The interpretation of each of the meta-parameters is given below:

component_type
This meta-parameter permits the basic generality of each Move, such that the only restriction on component_type is that it is identical for source and target.

source_packing
This meta-parameter permits the source to be unpacked, PACKED, or CRUNCHED independent of target_packing.

target_packing
This meta-parameter permits the target to be unpacked, PACKED, or CRUNCHED independent of source_packing.

move_error
This meta-parameter permits implementation flexibility, such that the method of signalling the error is not defined beyond that it is signaled.

move_r_to_l_move_l_to_r
This meta-parameter permits the description of both *Move_R_to_L* and *Move_L_to_R* in the procedure given above. The CASE on this meta- parameter selects the component movement semantics for each procedure.

MOVE Procedure Example

```
MODULE MoveExample;

IMPORT OtherModule;

EXPORT

   PROCEDURE ExampleProc;

IMPLEMENT

   TYPE
      IxType1 = 0..20;
      IxType2 = -3..17;

      Schema1 (L,H : IxType2) =
         ARRAY [L..H,IxType1] OF ImportedType;
      Schema2 (L,H : IxType1) =
         CRUNCHED ARRAY [L..H] OF ImportedType;

      Discrm1 = Schema1 (0,15);
      Discrm2 = Schema2 (5,10);

      Array1 = PACKED ARRAY [IxType1] OF ImportedType;
      Array2 = ARRAY [IxType2,IxType1] OF ImportedType;

   VAR
      DVar1 : Discrm1;
      DVar2 : Discrm2;
      AVar1 : Array1;
      AVar2 : Array2;

   PROCEDURE ExampleProc;
   VAR
      ix : Integer;

      BEGIN
      Move_L_to_R (10,DVar1,6,AVar2,0);
      FOR ix := 0 TO 9 DO {equivalent FOR loop}
         AVar2[ix] := DVar1[ix+6];

      Move_L_to_R (5,AVar2[5],-3,AVar1,-3);
      FOR ix := 0 TO 4 DO {equivalent FOR loop}
         AVar1[ix-3] := AVar2[5,ix-3];

      Move_R_to_L (6,DVar2,5,AVar1,5);
      FOR ix := 5 DOWNTO 0 DO {equivalent FOR loop}
         AVar1[ix+5] := DVar2[ix+5];
      END;

END.
```

7. XCall Mechanism

EXtermely Flexible Procedure *CALL* Mechanism

a. Background

Some system software needs to call routines whose identity and parameter list structure are COMPLETELY UNKNOWN until the system is executing. The unknown structure of the parameter list makes the strong type checking of CALL/FCALL inadequate for this SMALL class of system software. The predefined procedure *XCall* was included in BCG MODCAL to support ONLY this small class of system software. The portability of *XCall* is limited by the requirement that the system programmer provide for the (machine dependent) structuring of the parameter list. The use of this feature should be VERY limited.

b. Definition

The following procedure declaration describes the calling sequence and semantics of the *XCall* predefined procedure. The definition of a supporting predifined procedure, predefined types and a compiler option are described in a later section. Not all parameters or operations of *XCall* may be supported on all target machines (e.g. del_cnt for VISION), their use may be flagged.

Simply *XCall* checks the given preconditions. Copies a parameter list from a buffer built by the system software into the "parameter-area" of the target machine (e.g. the stack for the HP3000). Invokes the routine denoted by the procedural variable paramenter. The invoked routine may or may not clear/delete part of the "parameter-area". When the invoked routine completes (a machine dependent) part of the "parameter-area" may be copied into a buffer. Finally the "parameter-area" is cleared/deleted and *XCall* completes.

```
      PROCEDURE XCall  (
                proc_var  :  NullProc;
         ANYVAR in_buf    :  SU;
                in_cnt    :  ObjectSize;
                del_cnt   :  ObjectSize;
         ANYVAR out_buf   :  SU;
                out_cnt   :  ObjectSize
         )
      OPTION
         Inline
         Default_Parm  (
            in_buf   := NIL,
            in_cnt   := 0,
            del_cnt  := 0,
            out_buf  := NIL,
            out_cnt  := 0);

         BEGIN
         ASSERT (proc_var <> NIL                           , XCALL_err);
         ASSERT (in_cnt >= (del_cnt + out_cnt)             , XCALL_err);
         ASSERT ((in_cnt = 0)  OR (HaveOptVarParm(in_buf)) , XCALL_err);
         ASSERT ((out_cnt = 0) OR (HaveOptVarParm(out_buf)), XCALL_err);

         {COPY in_cnt SUs FROM in_buf INTO "PARAMETER-AREA"}

         {INVOKE proc_var}

         {COPY out_cnt SUs FROM "PARAMETER-AREA" INTO out_buf}

         {CLEAR/DELETE in_cnt - del_cnt SUs FROM "PARAMETER-AREA"}

         END;
```

c. Supporting Definitions

The *MakeRoutine* predefined procedure is intended to provide a mapping from a target machine dependent form to a target machine independent (language encapulated) form for procedural variables. If the mapping can not made "proc_var" is set to NIL.

```
        PROCEDURE MakeRoutine  (
           VAR proc_var    : NullProc;
               plabel_val : PLabel
           )
        OPTION
           Inline
           Type_Coerce_Parms  (
              proc_var := 'Similar_Form');

           BEGIN

           {MAP plabel_val INTO proc_var}

           END;
```

The following type PLabel is added to the foundation module for each target machine (operating system).

```
        TYPE
           PLabel = -32768 .. -1;      (*HP3000 - CST ext.*)
           PLabel = -32768 .. 32767;   (*HP3000 + CST ext.*)
           PLabel = MinInt .. MaxInt;  (*VISION*)
```

The type NullProc is any user procedural type variable with no parameters. For example:

```
        TYPE
           NullProc = PROCEDURE;
```

The XCALL_MACHINE compiler option denotes to the compiler and the system programmer which target machine's parameter passing conventions are used in any *XCalls*. This option must specify the same target machine name as the $TARGET_MACHINE compiler option did. This option may appear anywhere between tokens. It must appear and must specify a value other than 'NONE' before any *XCall* is encountered. It should also be used to "bracket" any code used to build parameter lists.

```
        $XCALL_MACHINE 'NONE'$       {default}
        $XCALL_MACHINE 'HP3000'$
        $XCALL_MACHINE 'VISION'$
```

d. XCall Example (HP3000)

```
TYPE
  NoParms = PROCEDURE;

VAR
  i,j,k  : INTEGER;
  buffer : ARRAY[1..3] OF INTEGER;

FUNCTION Min__Int (parm1,parm2 : INTEGER): INTEGER;
  EXTERNAL;

  BEGIN
  $XCall__MACHINE 'HP3000'$
  buffer[1] := 0; {function return space}
  buffer[2] := i;
  buffer[3] := j;

  $PUSH,TYPE__COERCION 'REPRESENTATION'$

  XCall (NoParms(Addr(Min__Int) ),
    buffer, 6, {pushed onto stack}
    4,        {EXIT 4 from Min__Int}
    k, 2);    {poped from stack}

  $POP,XCALL__MACHINE 'NONE'$
  END;
```

8. Multiple Heaps <u>NOT IMPLEMENTED</u>

However, routines can be written to manage MODCAL objects in a "heap-like" fashion, by manipulating offsets.

```
        _
       / \    _
      /   \_/ \
     /  THE    \
    /   HEAP    \
   ---------------
```

```
      _                              _
     / \    _                       / \    _
    /   \_/ \                      /   \_/ \
   /  other  \                    /  other  \
  /  Heap #1  \                  /  Heap #2  \
 ---------------                ---------------
```

```
      _                              _
     / \    _                       / \    _
    /   \_/ \                      /   \_/ \
   /  other                       /  other  \
  /  Heap #3   \                 /  Heap #4  \
 ---------------                ---------------
```

E. Standardized Modules

1. Portability Foundation

In order to assist the programmer in writing programs which will be portable between various machines, MODCAL must provide facilities for determining attributes of the targeted computer architecture. This section describes a set of predefined types, constants, and functions which will provide this function. These facilities are essentially a subset of the portability facilities found in Ada.

The cornerstone of these facilities is the notion of a storage unit (SU). A SU is the most useful addressable unit of storage on the target machine. All memory-related features of MODCAL (e.g., pointers, offsets, and objects) are defined in terms of SU's and many of the predefined functions work in terms of SU's.

In MODCAL/VCF integer is the default size for arithmetic expressions. No expression will use Longint arithmetic unless a variable or constant of Longint is in the expression. Thus it possible to have arithmetic traps on expressions involving only the integer type. Longint will only be supported on Vision. The assignment compatibility rules which apply to type integer also apply to type shortint and longint.

Note: LongInt constants MaxLongInt or MinLongInt or any constant which needs more than 32 bits of storage WILL NOT be available on the MODCAL/VCF cross compiler.

Floating point is not generally required for systems programming and so will not be addressed in this proposal.

Using the Sizeof function instead of absolute, hardcoded constants will aid in portability, since the values will automatically be adjusted when a program is recompiled for a different target machine. In addition to Sizeof, MODCAL/3000 and MODCAL/VCF have:

Bitsizeof(t) Gives the minimum number of bits required to represent a variable of type t.

Bitsizeof(t1, t2, ... ,tn) Gives the minimum number of bits required to represent a variable of type t1 with tag field values t2, ... , tn.

BitSizeof(s1, s2, ... ,sn) Gives the minimum number of bits required to represent a variable of the discriminated schema type select from the schema array s1 with discriminate values s2,...,sn.

SUsizeof (t)

SUsizeof (t1, t2, ... ,tn)

SUsizeof (s1, s2, ..., sn)

SUsizeof is similar to *Bitsizeof* but returns the storage size in SU units instead of bits.

The following are the predefined types and constants. The exact values of these types and constants are indicated below:

a. HP3000 Foundation

```
CONST
   MachineName = 'HP3000';
   Minint = -2147483648;       (* -2**31 *)
   Maxint =  2147483647;       (* (2**31)-1 *)
   MaxObjectSize = 32760;      (* maximum number of SUs in a HP 3000
                                  object *)

TYPE
   SU = -32768..32767;         (* A type requiring a SU of space *)
                               (* 16 bits on HP 3000 *)
   PointSU = ^SU;
   OffsetSU = OFFSET TO SU;
   ObjectSize = 0..MaxObjectSize;
   ObjectDelta = -MaxObjectSize..MaxObjectSize;

   ShortInt = -32768..32767; (* 16 bit, 2's complement integer *)
   Integer = Minint .. Maxint;

   Objectid = Shortint;
```

b. Vision Foundation

```
CONST
   MachineName = 'Vision';
   Minint = -2147483648;       (* -2**31 *)
   Maxint =  2147483647;       (* (2**31)-1 *)
   MaxObjectSize = Maxint;     (* maximum number of SUs in a Vision
                                  object *)
{the following 2 constants are not implemented in the cross compiler}
   Maxlongint = 9223372036854775807;  (* (2**63)-1 Vision only*)
   Minlongint = -9223372036854775808; (* -(2**63)  Vision only *)

TYPE
   SU = 0..255;                (* A type requiring a SU of space *)
                               (* 8 bits on VCF *)
   PointSU = ^SU;
   OffsetSU = OFFSET TO SU;
   ObjectSize = 0..MaxObjectSize;
   ObjectDelta = -MaxObjectSize..MaxObjectSize;

   ShortInt = -32768..32767; (* 16 bit, 2's complement integer *)
   Integer = Minint .. Maxint;
   LongInt = -Minlongint .. Maxlongint; (* VCF machine only *)
```

```
          Objectid = Integer;
```

2. Assembly Language

A set of predefined procedures will be provided, which when seen by the compiler, will emit instructions that are not normally generated, such as SINC, and DISP on the HP 3000, and PROBE on VCF. This set of predefined procedures is in the process of being determined. The list given here is for Version 5 of the Vision emulator only.

The normal method for accessing assembly language is to compile the code into a SPL procedure on the HP 3000, or using the assembler on Vision.

a. HP3000 Predefined Procedures

```
      TYPE
           Word = ShortInt;      { SizeOf = 2 }
           PAC = Any Packed Array of Char
                        or String

Function  GetDl : Word;
Function  GetQ : Word;
Function  GetS : Word;
Function  GetStatus : Word;
Procedure PutQ (Q_val : Word);     {NOT IMPLEMENTED}
Procedure PutStatus (Status_val: Word);
Procedure MoveBytesWhile (var Source,
                              Target: PAC;
                              Alpha, Numeric, UpShift: Boolean;
                          var Position: Word);
Function  ScanWhile (var Source: PAC;
                         TestChar, TermChar: Char;
                     var Position: Word) : Boolean;
Function  ScanUntil (var Source: PAC;
                         TestChar, TermChar: Char;
                     var Position: Word) : Boolean;
Function  CmpBytes (    Len: Word;
                    var Source,
                        Target: PAC
                   ) : Word;
```

b. Vision Predefined Procedures

```
TYPE
    Byte  = (op1, op2, op3); { SizeOf = 1 }
    Byte4 = Integer;         { SizeOf = 4 }
    Byte8 = LongInt;         { SizeOf = 8 }

Procedure Break;
Procedure CIS (    Channel: Byte;
                   Status: Byte
               var ConditionCode: Byte);
```

```
Procedure CmpC (    Length: Byte4;
                    Str1Ptr: Byte8;
                    Str2Ptr: Byte8;
                var Index: Byte4;
                var Result: Byte1);
Procedure CvLATVA (    Operand: Byte;
                   var GSVA: Byte8);
Procedure CvVATPP (    GSVA: Byte8;
                   var Physicalpage: Byte4);
Procedure Disable;
Procedure Disp;
Procedure Down (var Semaphore: Byte4);
Procedure Enable;
Function  GetQ: Byte8;
Procedure Halt_Inst;
Procedure Hash (    GSVA: Byte8;
                var HashIndex: Byte4);
Procedure IExit;
Procedure Interrupt (var OldIntVal: Byte);
Procedure IfC;
Procedure Idle;
Procedure ISwitch;
Procedure Launch (TCBA: Byte8;
                  TCBVA: Byte8);
Procedure MoveFSp4 (    Selector: Byte;
                    var Destination: Byte4);
Procedure MoveFSp8 (    Selector: Byte;
                    var Destination: Byte8);
Procedure MoveSema (    Operand: Byte4;
                    var Result: Byte4);
Procedure MoveTSp4 (Selector: Byte;
                    Source: Byte4);
Procedure MoveTSp8 (Selector: Byte;
                    Source: Byte8);
Procedure PDDel (Physpage: Byte4);
Procedure PDIns (Physpage: Byte4);
Function  Probe (Ring: Byte; Access: Byte;
                 Address: Byte8;
                 Lenght: Byte4): Boolean;
Procedure PsDb;
Procedure PsEb;
Procedure PurgeIB;
Procedure PurgeTLB;
Procedure RCl;
Procedure RDP (    Channel: Byte;
               var Data: ????
               var Length: Byte;
               var Result: Byte );
Procedure Restore_Registers (RegPtr: Byte8);
Procedure RIS (    Channel: Byte;
               var Status: Byte;
               var Result: Byte);
Procedure RSwitch;
Procedure Save_Registers (var RegPtr: Byte8);
Procedure SetIRT (Channel: Byte;
                  Intr_Pri_Level: Byte;
                  Parm: Byte4 );
Procedure SetQ_Exit (NewQ: Byte8);
Procedure SetQ_IExit (NewQ: Byte8);
Procedure SIS (    Channel: Byte;
                    Status: Byte;
               var Result: byte);
Procedure StartIO_Inst (SubChannel: Byte4;
                        ChannelProg: Byte8);
Procedure Stop;
Procedure Switch;
Function  TestA : Boolean;
Function  TestB : Boolean;
Function  TestDown (var Semaphore: Byte4): Boolean;
Function  TestOv : Boolean;
Function  TestRef (GVSP: Byte8): Boolean;
Function  TestSema (var Operand: Byte4;
                        Result: Byte4): Boolean;
Procedure Up (var Semaphore: Byte4);
Procedure VsimIn (    ReadCount: Byte4;
                  var ActualCount: Byte4;
                      BufferPtr: Byte8);
Procedure VsimOut (WriteCount: Byte4;
                   BufferPtr: Byte8);
Procedure WDP (    Channel: Byte;
                   Data: ????
               var Lenght: Byte);
```

The following Version 3 emulator predefined procedures are deleted from the ERS and the compiler:

```
IntOff
ReSetInt
IExit
CheckA
CheckB
EditTBL
EditIB
AsReset
GrpZero
CvLAtVP
CvVPtPP
```

Appendix A: Options and Directives

1. PROCEDURE/FUNCTION DEFINITION OPTIONS

Definition options specify optional attributes of a procedure or function (e.g. method of envoking it) as part of the routine head; the form is:

```
PROCEDURE proc_name (parm_name : type_name)
OPTION INLINE;
   BEGIN
   END;
```

the syntax is:

```
<routine_heading>
   -> <procedure_heading>  <defn_options> ?
   -> <function_heading>  <defn_options> ?

<defn_options>
   -> 'OPTION'  <defn_opt> +

<defn_opt>
   -> <option_id>  ( <option_spec_list>  |
                     <option_spec_value> ) ?

<option_id>
   -> <identifier>

<option_spec_list>
   -> '('  <opt_spec_assoc> LIST ','  ')'

<opt_spec_assoc>
   -> <spec_parm_id>  ':='  <expression>

<spec_parm_id>
   -> <identifier>

<option_spec_value>
   -> <integer>
```

Pascal Definition Options
   not a feature of Pascal

MODCAL Definition Options and Restrictions
(which <option_id>'s can be used together)

MODCAL Keyword	Parameter/Calling Convention	Optional Parameters	Linker Binding	Other
				{};
OPTION			Unresolved	{3};
OPTION		Default_Parms{1}		{};
OPTION		Default_Parms{1}	Unresolved	{3};
OPTION	Inline			{4};
OPTION	Inline	Default_Parms{1}		{4};
OPTION	Gateway			{5};
OPTION	Gateway		Unresolved	{5};
OPTION	Gateway	Default_Parms{1}		{5};
OPTION	Gateway	Default_Parms{1}	Unresolved	{5};
OPTION	Extensible_Gateway{2}			{5};
OPTION	Extensible_Gateway{2}		Unresolved	{5};
OPTION	Extensible_Gateway{2}	Default_Parms{1}		{5};
OPTION	Extensible_Gateway{2}	Default_Parms{1}	Unresolved	{5};
OPTION	Interrupt_Parms{1}			{3,6};
OPTION	Interrupt_Parms{1}		Unresolved	{3,6};

{1}	<opt_defn> -> <option_id> <option_spec_list>
{2}	<opt_defn> -> <option_id> <option_spec_value> ?
{3}	routine is declared on level 1
{4}	routine is non-recursive
{5}	routine is declared in EXPORT part
{6}	PROCEDUREs only

note : Uncheckable__ANYVAR can be used with any other definition option

a. OPTION Inline

Definition option Inline denotes a procedure or function that will be expanded inline wherever it is invoked. This macro-like expansion removes most procedure call overhead and increases the amount of object code generated. This expansion is performed so as to retain "Call-By-Reference" where a simple macro expansion would result in "Call-By-Name" (i.e. variable parameters work the same with OPTION Inline). Inline procedures or functions cannot invoke themselves or any other mutually recursive inline procedures or functions. **If an inline procedure or function contains any non-inline procedures or functions $Private__Proc Off$ MUST BE specified for them.**

b. OPTION Gateway

Definition option Gateway denotes a procedure or function that is a gateway into the MODCAL environment. That is, Gateway procedures or functions are MODCAL procedures or functions that must support the system defined calling conventions. In other words, Gateway is used to denote intrinsics that can be called from any programming language. This definition option can be specified only on level one procedures or functions.

c. OPTION Extensible_Gateway

Definition option Extensible_Gateway denotes a procedure or function that is a gateway into the MODCAL environment and that has an extensible parameter list. That is the parameter list has a fixed number of non-extension parameters and any number of optional extension parameters. The integer value part of this definition option specifies the number of non-extension parameters. This definition option can be specified only on level one procedures or functions. (see *HaveExtension* function)

NOTE: This option will be accepted by MODCAL/3000 but it will have several special properties. The specification of extension parameters is not supported, due to 3000 parameter stacking conventions. Code generated for one of these Extensible__Gateway routines expects an EXTERNAL SPL VARIABLE (in Pascal/MODCAL) style bit map. This implies that they should be specified as OPTION VARIABLE EXTERNAL (in SPL) to BUILDINT.PUB.SYS. The bit map is checked just before any data references. Any omitted optional parameters are supplied (see Option Default__Parms) and any omitted required parameter causes an *Escape*. This checking delay permits the first statement in a routine to be a TRY, so missing parameter escapes can be handled.

d. OPTION Interrupt_Parms

Definition option Interrupt_Parms denotes a procedure that has its parameter list allocated AS SPECIFIED BY the <option_spec_list> part of this option. The <spec_parm_id> denotes the formal parameter and the constant <expression> denotes the "activation base relative" offset (on VCF the Q base register relative SU, byte, offset) of that parameter. This option can be specified only on level one procedures. These procedure have their local variables allocated so as to not overlay any parameters. The consistency of the parameter offsets IS NOT CHECKED. No procedure defined with the option may appear in any procedure statement. No variable of a procedural type defined with this option may appear in any *Call*. Only the standard activation record or stack marker is deleted when this procedure exits.

NOTE: FORM OF PROCEDURE EXIT IS SUBJECT TO REVISION!!!

NOTE: MODCAL/3000 WILL TREAT THIS OPTION AS AN NO OP!!!

e. OPTION Default_Parms

Definition option Default_Parms denotes which parameters may be omitted in any actual parameter list for this procedure or function. This is specified by the parameter name appearing as the <spec_parm_id> in the <option_spec_list> syntax, with an assignment compatible constant <expression>. If the actual parameter is omitted the compiler suppilies the default value for the formal parameter corresponding to the position of the omitted actual parameter. This requires that the compiler supply all the parameters defined in the formal parameter list for every call. Thus any formal parameter that does not appear as a <spec_parm_id> in this definition option is required and can not be omitted from ANY actual parameter list. The only default value permitted for a variable (VAR), any variable (ANYVAR), procedure, or function parameter is a SPECIAL CASE USE of NIL. (see *HaveOptVarParm* function)

f. OPTION Unresolved

Definition option Unresolved denotes a procedure or function that is left unresolved by both the segmenter/linker and the loader. The resolution of the symbolic name to its reference part is delayed until the procedure or function is used (e.g. LOADPROC (); PCAL 0 / CALLPROC (); CALLV). The suggested way to use this kind of procedure or function is to use *Addr* to determine if it can be resolved (NIL will be returned if it is not). This definition option can be specified only on level one procedures or functions.

g. OPTION Uncheckable_ANYVAR NOT IMPLEMENTED

Defintion option Uncheckable__ANYVAR indicates the no "phantom" size-of parameter is to be created for the ANYVAR parameters in the formal parameter list. It is an error to specify option Uncheckable__ANYVAR if there are no ANYVAR formal parameters. It is not possible to perform checking of ANYVAR parameter accesses if this option is specified. The use of this definition option greatly increases the danger of undetected errors in the use of ANYVAR parameters. (see Section D)

h. *HaveExtension* and *HaveOptVarParm* Functions

The predefined boolean function *HaveExtension* indicates for a formal parameter name of the surrounding scope whether an actual extension parameter was supplied, either explicitly or by default (TRUE) or the formal parameter appears after the last supplied actual parameter in the parameter list (FALSE). If the formal parameter name is not an extension parameter a compile time error is issued.

The predefined boolean function *HaveOptVarParm* indicates for a formal parameter name of the surrounding scope whether an actual parameter was supplied (TRUE) or it was defaulted to NIL (FALSE). If the formal parameter name is not an optional variable, any variable, procedure, or function parameter a compile time error is issued.

i. Definition Option Example

```
$TARGET_MACHINE 'VISION'$

   PROCEDURE UseNewFuncs  (
      VAR Parm1 : Type1;
      VAR Parm2 : Type2 )
   OPTION
      Extensible_Gateway 1
      Default_Parms (
         Parm1 := NIL,
         Parm2 := NIL );
      BEGIN
      IF HaveExtension (Parm2) THEN
         IF HaveOptVarParm (Parm1) THEN
            IF HaveOptVarParm (Parm2) THEN
               (* CALL 1 *)
            ELSE
               (* CALL 3 *)
         ELSE
            IF HaveOptVarParm (Parm2) THEN
               (* CALL 2 *)
            ELSE
               (* CALL 4 *)
      ELSE
         IF HaveOptVarParm (Parm1) THEN
            (* CALL 5 *)
            Assert (NOT HaveOptVarParm (Parm2), 3)
         ELSE
            (* CALL 6 *)
            Assert (NOT HaveOptVarParm (Parm2), 4);
      END;

     { It is assumed that the following calls to "UseNewFuncs"
       were compiled AFTER its parameter list was extended by
       the addition of "Parm2" }
   (* CALL 1 *)  UseNewFuncs (Var1,Var2);
   (* CALL 2 *)  UseNewFuncs (,Var2);
   (* CALL 3 *)  UseNewFuncs (Var1);
   (* CALL 4 *)  UseNewFuncs ();

     { It is assumed that the following calls to "UseNewFuncs"
       were compiled BEFORE its parameter list was extended by
       the addition of "Parm2" }
   (* CALL 5 *)  UseNewFuncs (Var1);
   (* CALL 6 *)  UseNewFuncs ();
```

2. PROCEDURE/FUNCTION DIRECTIVES

Pascal Procedure/Function Directives
- *EXTERNAL
  - external SPL 3000 only
  - external SPL VARIABLE 3000 only
  - external FORTRAN
  - external COBOL
- *FORWARD
- *INTRINSIC

MODCAL Procedure/Function Directives
- *EXTERNAL
  - external ASSEMBLER Vision only NOT IMPLEMENTED

a. External

Directive External indicates that the procedure or function will be compiled/assembled by another language processor (default is Pascal). The MODCAL language processor assumes that the procedure or function will use the system defined calling conventions.

NOT IMPLEMENTED

	EXTERNAL			
		any	SPL	ASSEMBLER
Gateway	yes	Error	Error	Error
Extensible_Gateway	yes	Error	Error	Error
Unresolved	yes	yes	yes	yes
Inline	Error	Error	Error	some day?
Default_Parms	yes	yes	yes	yes
Interrupt_Parms	yes	Error	yes	yes
Uncheckable_ANYVAR	yes	yes	yes	yes

{ any IN [COBOL,FORTRAN,SPL VARIABLE] }

b. Forward

Directive Forward indicates that the procedure or function body occurs somewhere later in the compilation unit. This directive permits mutually refering procedures and functions.

NOT IMPLEMENTED

	FORWARD
Gateway	yes
Extensible_Gateway	yes
Unresolved	Error
Inline	Error
Default_Parms	yes
Interrupt_Parms	yes
Uncheckable_ANYVAR	yes

c. Intrinsic

Directive Intrinsic indicates that the procedure or function declaration is contained in the system intrinsic file.

NOT IMPLEMENTED

	INTRINSIC
Gateway	Error
Extensible_Gateway	Error
Unresolved	yes
Inline	Error
Default_Parms	yes
Interrupt_Parms	Error
Uncheckable_ANYVAR	yes

3. COMPILER OPTIONS

Compiler options specify compilation controls (e.g. features allowed, disabling run-time checking); the form is:

```
$STANDARD_LEVEL 'BCG_MODCAL'$
PROGRAM prog_name;
   PROCEDURE proc_name
      $EXEC_PRIVILEGE 0$
      (parm_name : type_name);
      .
      .
      $BEGIN_CHANGE '001601'$
      .
      .
      $END_CHANGE '001601',PAGE$
```

```
Pascal Compiler Options
   Compilation Unit Options
     *COPYRIGHT
         copyright ''
     *EXTERNAL
     *GLOBAL
     *HEAP_COMPACT
         heap_compact ON
         heap_compact OFF
     *HEAP_DISPOSE
         heap_dispose ON
         heap_dispose OFF
     *SET
         set ''
     *SUBPROGRAM
         subprogram
         subprogram ''
     *USLINIT

   Routine Head Options
     *ALIAS
         alias ''
      SYMDEBUG
         symdebug ON
         symdebug OFF

   Any Token Options, Deferred
     *CHECK_ACTUAL_PARM
         check_actual_parm 0..3
     *CHECK_FORMAL_PARM
         check_formal_parm 0..3
     *CODE
         code ON
         code OFF
     *CODE_OFFSETS
         code_offsets ON
```

```
        code_offsets OFF
    *LINES
        lines 20..MAXINT
    *LIST_CODE
        list_code ON
        list_code OFF
    *PRIVATE_PROC
        private_proc ON
        private_proc OFF
    *SEGMENT
        segment ''
    *SPLINTR
        splintr ''
     STATS
        stats ON
        stats OFF
    *TABLES
        tables ON
        tables OFF
    *TITLE
        title ''
    *XREF
        xref ON
        xref OFF

  Any Token Options, Immediate
    *ANSI
        ansi ON
        ansi OFF
    *ASSERT_HALT
        assert_halt ON
        assert_halt OFF
    *ELSE
        else
    *ENDIF
        endif
    *FONT
        font ''
    *IF
        if ''
    *INCLUDE
        include ''
    *LIST
        list ON
        list OFF
    *PAGE
    *PARTIAL_EVAL
        partial_eval ON
        partial_eval OFF
    *RANGE
        range ON
        range OFF
    *SKIP_TEXT
        skip_text ON
        skip_text OFF
    *STANDARD_LEVEL
        standard_level 'ANSI'
        standard_level 'HP STANDARD'
        standard_level 'HP3000'
    *WIDTH
        width 10..128

MODCAL Compiler Options
   Compilation Unit Options
    *ACD_VERSION
        acd_version 3..5
     CRUNCH_PACKING
        crunch_packing ON
        crunch_packing OFF
     MULTIPLE_HEAPS
        multiple_heaps ON
        multiple_heaps OFF
     INITIAL_EXEC_PROBE
        initial_exec_probe 0..MAXINT
     OBJECT_DEFINITION
        object_definition ON
        object_definition OFF
     OPTION_LOGGING
        option_logging ON
        option_logging OFF
     SCHEMA_ARRAYS
        schema_arrays ON
        schema_arrays OFF
    *TARGET_MACHINE
        target_machine 'HP3000'
        target_machine 'VISION'

   Routine Head Options
    *CALL_PRIVILEGE
        call_privilige 0..3
     DYNAMIC_SIDE_EFFECTS_OK
        dynamic_side_effects_ok ON
        dynamic_side_effects_ok OFF
    *EXEC_PRIVILEGE
        exec_privilige 0..3
     PROBE_RESOLUTION  (see DEBUG)
        probe_resolution 'NO PROBES'
        probe_resolution 'ROUTINE'
        probe_resolution 'DECISION_PATH'
        probe_resolution 'STATEMENT'
        probe_resolution 'OPERATION'
     STATIC_SIDE_EFFECTS_OK
        static_side_effects_ok ON
        static_side_effects_ok OFF
     UNDEFINED_VAR_CHECKING
        undefined_var_checking ON
        undefined_var_checking OFF
```

```
Any Token Options, Deferred
   BEGIN_CHANGE
      begin_change ''
   CHANGE_INFO_WIDTH
      change_info_width 4..30
   END_CHANGE
      end_change ''
   OBJECT_SELECTION
      object_selection ON
      object_selection OFF
   PROBE_INSERTION
      probe_insertion ON
      probe_insertion OFF
   SEARCH
     search ''
   TABULATE_CHANGE_INFO
      tabulate_change_info ON
      tabulate_change_info OFF
  *TYPE_COERCION
      type_coercion 'NONE'
      type_coercion 'STRUCTURAL'
      type_coercion 'REPRESENTATION'
      type_coercion 'STORAGE'
      type_coercion 'NONCOMPATIBLE'

Any Token Options, Immediate
   INDEX_CHECKING
      index_checking ON
      index_checking OFF
   OPEN_SCOPE_CHECKING
      open_scope_checking ON
      open_scope_checking OFF
  *POP
  *PUSH
   REFERENCE_CHECKING
      reference_checking ON
      reference_checking OFF
  *STANDARD_LEVEL
      standard_level 'HP_MODCAL'
      standard_level 'BCG_MODCAL'
   SUBRANGE_CHECKING
      subrange_checking ON
      subrange_checking OFF
   SCHEMA_CHECKING
      schema_checking ON
      schema_checking OFF
   TAG_CHECKING
      tag_checking ON
      tag_checking OFF
   UNDEFINED_PARM_CHECKING
      undefined_parm_checking ON
      undefined_parm_checking OFF
  *XCALL_MACHINE
      xcall_machine 'NONE'
xcall_machine 'HP3000'
xcall_machine 'VISION'
```

(*) denotes feature operational in current release MODCAL/3000

a. ADC_Version

This compiler option is used to inform the compiler which version of the Vision ACD is be used, so that the correct set of assembly instructions is made availible.

b. Alias

This compiler option substitutes an alias as an external name for a procedure or function. See the Pascal/3000 reference manual.

c. Ansi

This compiler option causes all non ANSI Standard Pascal features to be flagged. The default setting is OFF. See the Pascal/3000 reference manual.

d. Assert_Halt

This compiler option specifies that when an assertion fails, the program will terminate. The default setting is OFF. See the Pascal/3000 reference manual.

e. Begin_Change NOT IMPLEMENTED

This compiler option specifies the identifying string to be made the most recent in the change history of the source. The change history is displayed to the right of the source line with the most resent change closest to the source. See Tabulate__Change__Info option.

f. Call_Privilege

This compiler option specifies the least privilege (largest numerical value) that any procedure or function may be executing at when it invokes this procedure or function. This compiler option provides the functionality of OPTION UNCALLABLE in SPL/3000 (HP3000: 0 -> Uncallable, 1 -> Callable). Default is least privilege on target machine (HP3000: 1, Vision: 3).

g. Change_Info_Width NOT IMPLEMENTED

This compiler option specifies the number of characters displayed in each column of the change history. Default is 8.

h. Check_Actual_Parm

This compiler option specifies the level of checking the Segmenter will perform when MODCAL calls a procedure or function. The default setting is level 3. See the Pascal/3000 reference manual.

i. Check_Formal_Parm

This compiler option specifies the level of checking the Segmenter will perform when a procedure or function is called. The default setting is level 3. See the Pascal/3000 reference manual.

j. Code

This compiler option casues object code to be generated at the end of each procedure, function or outer block. The default setting is on. See the Pascal/3000 reference manual.

k. Code_Offsets

This compiler option causes a table of P locations to be printed for each statement. The default setting is OFF. See the Pascal/3000 reference manual.

l. Copyright

This compiler option inserts a copyright notice and the specified name in the USL file. See the Pascal/3000 reference manual.

m. Crunch_Packing NOT IMPLEMENTED

Crunch_Packing ON tells the compiler that the Crunch Packing feature of MODCAL will be used in the source. OFF tells the compiler that CRUNCHED will not be used; CRUNCHED will then be flagged as an error. Default is ON.

n. Dynamic_Side_Effects_Ok NOT IMPLEMENTED

This compiler option turns ON and OFF the ability to make modifications to the variables referenced by reference types. Default is ON.

o. Else

This compiler option is used in conditional compilation. If the corresponding $IF expression is FALSE, then the source between the $ELSE and the corresponding $ENDIF will be compiled.

p. EndIf

This compiler option is used in conditional compilation. $ENDIF signals the end of a $IF block or a $ELSE block.

q. End_Change <u>NOT IMPLEMENTED</u>

This compiler option specifies the (previously defined) identifying string that is to be removed from the change history. If the identifying string appear more than once the most recent is removed.

r. Exec_Privilege

This compiler option specifies the privilege at which this procedure or function will execute. This compiler option provides the functionality of OPTION PRIVILEGED in SPL/3000 (HP3000: 0 -> Priv-Mode, 1 -> User-Mode). Default is least privilege on target machine (HP3000: 1, Vision: 3).

s. External

This compiler option is used with the option Global to permit separate compilation of procedures and functions. See Pascal/3000 reference manual.

t. Font

This compiler option is used to change the primary and alternate character set used in the listing file (for use with 2680A laser printer) The string parameter is composed of 2 integers separated by a comma, the first is the number of the new primary set, and the second is the number of the new secondary set. Note: the switch between the character set can be done only within comments.

u. Global

This compiler option is used with the option External to permit separate compilation of procedures and functions. See Pascal/3000 reference manual.

v. Heap_Compact

This compiler option causes free areas in the heap to be combined. The default setting is OFF. See the Pascal/3000 reference manual.

w. Heap_Dispose

This compiler option permits disposed areas in the heap to be reallocated. The default setting is OFF. See the Pascal/3000 reference manual.

x. If

This compiler option is used in conditional compilation. If the expression in the string following the $IF evaluates to the logical value TRUE, then the source between the $IF and the corresponding $ENDIF or $ELSE will be compiled. If the expression is FALSE, then the source will be skipped. The expression may include the AND, OR, and NOT operators and parenthesis. $IF may be nested to a depth of 16.

y. Include

This compiler option permits the inclusion of another file which the compiler will process as source code. See the Pascal/3000 reference manual.

z. Index_Checking NOT IMPLEMENTED

This compiler option turns ON and OFF the generation of checking code for array indexing operations. Default is ON.

aa. Initial_Exec_Probe NOT IMPLEMENTED

Not yet fully defined.

ab. Lines

This compiler option specifies the number of lines that will appear on a single page of the listing. The default setting is 55. See the Pascal/3000 reference manual.

ac. List

This compiler option turns ON and OFF the compiler's listing. The default value is ON. See the Pascal/3000 reference manual.

ad. List_Code

This compiler option causes a mnemonic listing of the object code generated to appear at the end of each procedure, function, or outer block. The default setting is OFF. See the Pascal/3000 reference manual.

ae. Multiple_Heaps <u>NOT IMPLEMENTED</u>

Multiple__Heaps ON tells the compiler that the Multiple Heap feature of MODCAL will be used in the source. OFF tells the compiler that Multiple Heaps will not be used; any use of multiple heap features will then be flagged as an error. Default is OFF. Multiple heaps will not be available initially.

af. Object_Definition <u>NOT IMPLEMENTED</u>

Object__Definition ON tells the compiler that Object features will be used in the source. OFF tells the compiler that Objects will not be used; any use of object features will then be flagged as an error. Default is ON.

ag. Object_Selection *(will be implemented)*

This compiler option turns ON and OFF special code generation to optimize the access of the object most recently selected in a WITHOBJECT statement (e.g. on HP3000 calls to ExchangeDB). Default is ON.

ah. Open_Scope_Checking <u>NOT IMPLEMENTED</u>

This compiler option turns ON and OFF generation of checking code for open scope checking. The default value is ON.

ai. Option_Logging <u>NOT IMPLEMENTED</u>

This compiler option turns ON and OFF the generation of a statement by statement log file of the compiler options in effect. The default setting is ON.

aj. Page

This compiler option causes the compiler listing to a line printer to perform a page eject and start a new page. See the Pascal/3000 reference manual.

ak. Partial_Eval

This compiler option controls the way conditional (BOOLEAN) expressions are evaluated. Default is ON. See the Pascal/3000 reference manual.

al. Pop

This compiler option restores the compile option to the state before the last Push.

1. Compiler Options Unaffected by Pop

ACD__Version
Alias
Call__Privilege
Copyright
Exec__Privilege
External
Font
Global
Include
Page
Profile__Cnt
Push
Segment
Skip__Text
Splintr
Target__Machine
Title

am. Private_Proc

This compiler option is used to make (ON) nested procedures and functions (level > 1) private to the RBM containing the surrounding level 1. If it is OFF each procedure or function is placed in a separate RBM, and must have a name unique at 15 characters. Default is ON. See the Pascal/3000 reference manual.

an. Probe_Insertion NOT IMPLEMENTED

This compiler option turns ON and OFF the generation of software probes without affecting their syntactic resolution. Default is OFF. This option is set ON any time the Probe__Resolution option is specified.

ao. Probe_Resolution NOT IMPLEMENTED

This compiler option specifies the syntax unit resolution of software probes inserted into the code generated. Default is that no probes are inserted.

ap. Push

This compiler option saves the current setting of options in an option set. No options are changed.

1. Compiler Options Unaffected by Push

ACD_Version
Alias
Call_Privilege
Copyright
Exec_Privilege
External
Font
Global
Include
Page
Pop
Profile_Cnt
Segment
Skip_Text
Splintr
Target_Machine
Title

aq. Range

This compiler option causes range checking code to be generated. The default value is ON. See the Pascal/3000 reference manual.

1. Compiler Options Implied by $Range ON$

$Index_Checking	**ON$**
$Raference_Checking	**ON$**
$Subrange_Checking	**ON$**

2. Compiler Options Implied by $Range OFF$

$Index_Checking	**OFF$**
$Raference_Checking	**OFF$**
$Subrange_Checking	**OFF$**
$Open_Scope_Checking	**OFF$**
$Schema_Checking	**OFF$**
$Tag_Checking	**OFF$**
$Undefined_Parm_Checking	**OFF$**

ar. Reference_Checking *(will be implemented)*

This compiler option turns ON and OFF generation of checking code for references, i.e. pointer and object, offsets. The default value is ON.

as. Schema_Arrays <u>NOT IMPLEMENTED</u>

Schema_Arrays ON tells the compiler that Schema Array features will be used in the source. OFF tells the compiler that Schema Arrays will not be used; any use of schema array features will then be flagged as an error. Default is OFF.

at. Schema_Checking <u>NOT IMPLEMENTED</u>

This compiler option turns ON and OFF generation of checking code for schema arrays. The default value is ON.

au. Search <u>NOT IMPLEMENTED</u>

The string literal parameter for Search names external libraries to be searched when satisfying import lists. The libraries are searched in the order listed in the literal. This option overrides all prior Search options. Exactly how Search works is implementation defined. It has not yet been defined for MODCAL/VCF or MODCAL/3000.

av. Segment

This compiler option specifies the name for the current segment. The default segment name is Seg'. See the Pascal/3000 reference manual.

aw. Set

This compiler option is used in conditional compilation. $SET specifies all identifiers to be used as flags in the conditional compilation of the soucre. (See $IF.) The string which follows the $SET option contains a list consisting of an identifier, an '=', and the value of the identifier, either TRUE or FALSE. Each element is separated by a comma. (e.g. $SET 'Red=TRUE,Blue=FALSE'$)

ax. Skip_Text

This compiler option causes the compiler to ignore all subsequent source code, including any compiler options, until Skip_Text is turned OFF. See the Pascal/3000 reference manual.

ay. Splintr

This compiler option specifies which SPL intrinsic file to search for a procedure or function declared with the INTRINSIC directive. The default file is Splintr.pub.sys. See the Pascal/3000 reference manual.

az. Standard_Level

This compiler option specifies which syntax and semantics to accept. If the source code contains a feature which is not in the current langage level, the compiler will issue a warning. See the Pascal/3000 reference manual.

ba. Static_Side_Effects_Ok NOT IMPLEMENTED

This compiler option turns ON and OFF the ability to make modifications to nonlocal variables. Default is ON.

bb. Stats NOT IMPLEMENTED

This compiler option produces statistical information about the compilation. The default setting is OFF.

bc. Symdebug NOT IMPLEMENTED

This compiler option inserts symbolic debugging information into the object code. The default setting is OFF.

bd. Subprogram

This compiler option permits compilation of a subset of level 1 procedures or functions.

be. Subrange_Checking NOT IMPLEMENTED

This compiler option turns ON and OFF generation of checking code for subrange violations. The default value is ON.

bf. Tables

This compiler option causes an idendifier map to be produced at the end of each procedure, function or outer block. The map includes the class, type, and address or constant value for each identifier. The default setting is OFF. See the Pascal/3000 reference manual.

bg. Tabulate_Change_Info NOT IMPLEMENTED

This compiler option cause change information to appear in the right margin of the listing. The Begin__Change and End__Change options are not listed while Tabulate__Change__info is ON. The default setting is ON. See Begin__Change option.

bh. Tag_Checking *(will be implemented)*

This compiler option causes checking code for references to fields in the variant part of records to be checked against tag fields. The default setting is OFF.

bi. Target_Machine

This compiler option specifies which target machine to generate code for.

bj. Title

This compiler option specifies the string to be printed next to the page number on subsequent pages of the listing. See the Pascal/3000 reference manual.

bk. Type_Coercion

This compiler option specifies the level of type coercion to allow. The default setting is none.

bl. Undefined_Parm_Checking <u>NOT IMPLEMENTED</u>

This compiler option turns ON and OFF the generation of checking code for references to optional parameters. Default is OFF.

bm. Undefined_Var_Checking <u>NOT IMPLEMENTED</u>

This compiler option turns ON and OFF the generation of checking code for references to local variables not yet assigned to. Default is OFF.

bn. UslInit

This compiler option causes the USL file to be initialized to empty. See the Pascal/3000 reference manual.

bo. Width

This compiler option specifies the number of columns which the compiler will process from each input record. See the Pascal/3000 reference manual.

bp. XCall_Machine

See eXtremely flexible procedure CALL mechanism section.

bq. Xref

This compiler option causes a cross reference listing to be produced for each procedure, function or outer block. The default setting is OFF. See the Pascal/3000 reference manual.

Appendix B: Data Types and Representations

1. MODCAL Data Types

```
Simple Types                                            |X| | | |X| |
   Ordinal Types                                        |X| | | |X| |
      Integer Based Types    - integer                  |X| | | |X| |
      Character Based Types - char                      |X| | | |X| |
      Enumerated Based Types                            |X| | | |X| |
         Predefined - boolean                           |X| | | |X| |
         User Defined                                   |X| | | |X| |
   Real Types                                           |X| | | |X| |
      Single Binary Flt. Pt. - real                     |X| | | |X| |
      Double Binary Flt. Pt. - longreal                 |X| | | |X| |
Reference Types                                         |X| | | | |X|
   Routine Reference Types                              |X| | | | |X|
      Procedural Types - PROCEDURE                      |X| | | | |X|
      Functional Types - FUNCTION :                     |X| | | | |X|
   Data Reference Types                                 |X| | | | |X|
      Pointer Types - ^                                 |X| | | | |X|
      Object Types  - %                                 |X| | | | |X|
      Offset Types  - OFFSET TO                         |X| | | | |X|
Structured Types                                        |X|?|?|?|X| |
   Simple Structured Types                              |X| | | |X| |
      Record Types - RECORD CASE OF END                 |X| | | |X| |
      Array Types  - ARRAY [] OF                        |X| | | |X| |
      Set Types    - SET OF                             |X| | | |X| |
   Descripted Structured Types                          |X|?|?|?|X| |
      Nonimbedded Descriptor Types                      |X|X| | |X| |
         Schema Arrays        - () = ARRAY [] OF        |X|X| | |X| |
         Discriminated Schemas - ()                     |X|X| | |X| |
      Imbedded Descriptor Types                         |X| |?|?|X| |
         File Types   - FILE OF                         |X| |X| |X| |
         String Types - STRING []                       |X| | |X|X| |
                                                         + + + + + +
                                                         + + + + + +
                                                         + + + + + +
```

2. MODCAL Data Representation

```
                                                         + + + + + +
Reference Part                                           + + + + + +
   Location Information ---------------------------------+ + + + + +
   Nonimbedded Descriptor Information ---------------------+ + + + +
Data Value Part                                              + + + +
   Imbedded Descriptor Information                           + + + +
      Active Component Position Information -----------------+ + + +
      Valid Component Count Information -----------------------+ + +
   Value Bit Vector -------------------------------------------+ +
   Referenced Item Reference Part -------------------------------+
```

a. Location Information

The location information portion of the reference part of a data item describes how to locate the data value part of the data item. In its most generalized form this is an object offset pair or a static-link STT-entry pair.

b. Nonimbedded Descriptor Information

The nonimbedded discriptor information portion of the reference part of a data item describes what the discriminant_values associated with that data item are. This can be viewed as dope vector information.

c. Active Component Position Information

The active component position information portion of the data value part of a data item describes which component of a sequence of data items is currently contained in the value bit vector portion of the data value part. This portion may contain additional information.

d. Valid Component Count Information

The valid component count information portion of the data value part of a data item discribes the number of valid components currently contained in the value bit vector portion of the data value part.

e. Value Bit Vector

The value bit vector portion of the data value part of a data item is the actual bit vector representing the value of the data item. The value bit vector portion of structured data items contains the value bit vector protions of all of the component data items plus any "padding" bits.

f. Referenced Item Reference Part

The referenced item reference part portion of the data value part of a data item contains the refernce part of the referenced (pointed to) data item. This portion contains all the information needed to locate and access the referenced (pointed to) data item.

Appendix C: Type Compatibility

1. Pascal Compatible Type Compatibilities

```
      Implicit Type Conversion for assignment

   Identical Types    (T1 <=> T2)
      if either
       (1) T1.identifier = T2.identifier
       (2) TYPE T1.identifier = T2.idenitifier;
           appears in a declaration part

   Compatible Types    (T1 <= T2)
      if any
       (1) T1 and T2 are Identical Types
       (2) T1 and T2 are subranges of T3, or
           T1 is a subrange of T2, or
           T2 is a subrange of T1
       (3) TYPE T1 = SET OF T3; T2 = SET OF T4;,
           and T3 and T4 are Compatible Types,
           and T1.packing = T2.packing
       (4) T1 and T2 are PAC types with the same length or if a
           value of either T1 or T2 is a literal whose length
           <= the length of the other type which is a literal
           or PAC, the literal is extended on the right with
           blanks to reach a compatible type.
       (5) T1 is a string type and T2 is a string type
           or literal.
       (6) T1 and T2 are both Floating Point Types

   Assignment Compatible Types    (T1 <= T2)
       A value of type T2 is assignment compatible with a
       type T1 if any of the statements that follow are true
       and the restrictions following them are observed.

       (1) T1 and T2 are compatible types which are neither
           file types nor structured types which (recursively)
           contain file types.
       (2) T1 is a Real Type,
           and T2 is an Integer Type
       (3) T1 is a procedure type and T2 is a procedure
           with a parameter list congruent (as defined
           in HP Pascal) to the parameter list of T1
       (4) T1 is a function type and T2 is a function
           with a parameter list congruent to the
           parameter list of T1 and result type identical
           to the result type of T1

       The following rules must hold:
       (1) If T1 and T2 are Ordinal Types,
           then the value of type T2 must be in
```

interval specified by T1

(2) If T1 and T2 are Set Types, then all members of the value of type T2 must be in the interval of the base type of T1

(3) If T1 is a string type, and T2 is any string type or string literal, the length of the value of T2 must be <= the maximun length of T1. The length of the variable of type T1 is set to the length of the value of type T2.

(4) If T1 is a procedure or function type, then value of T2 must have the same or wider scope than the variable or parameter of type T1 being assigned to.

2. Pascal Incompatible Type Compatibilities

Explicit Type <u>Coercion</u> required for assignment

Structurally Compatible Types (T1 <=> T2)

Same number, order, and packing of Structurally Compatible Typed components (1 for 1 mapping)

Two types T1 and T2 are structurally compatible if any of the following are true:

(1) T1 and T2 are identical types

(2) T1 and T2 are ordinal types and the ORD(minimum value of T1)=ORD(minimum value of T2) and the ORD(maximum value of T1)=ORD(maximum value of T2)

AND

T1 & T2 base types are user enumerated types,
OR T1 & T2 base types are integer,
OR T1 & T2 base types are boolean,
OR T1 & T2 base types are char.

(3) T1 and T2 are types of a procedure. Two procedures have structurally compatible types if they have the same number of parameters and each corresponding parameter has the same form (Var or value) and the type of each corresponding parameter is structurally compatible

(4) T1 and T2 are types of a function. Two functions have structurally compatible types if they have the same number of parameters and each corresponding parameter has the same form (Var or value) and the type of each corresponding parameter is structurally compatible and the results types of both functions are structurally compatible

(5) T1 and T2 are data reference types of identical form (i.e. both are either pointer, object, offset) which reference data with structurally compatible types.

(6) T1 and T2 are record types with the same packing and each field in T1 is structurally compatible with the corresponding field in T2 and the number of fields in T1= the number of fields in T2
(7) T1 and T2 are array types with identical packing and structurally compatible components and index types
(8) T1 and T2 are set types with identical packing and structurally compatible base types
(9) T1 and T2 are file types with structurally compatible component types
(10) T1 and T2 are schema types with structurally compatible index types and structurally compatible component types and identical packing
(11) T1 and T2 are string types with identical maximum lengths

Representation Size Compatible Types (T1 <= T2)
BitSizeOf(T1) = *BitSizeOf*(T2)

Storage Size Compatible Types (T1 <= T2)
SUSizeOf(T1) >= *SUSizeOf*(T2)

NonCompatible Types (T1 <=> T2)
(1) *SizeOf*(T1) < *SizeOf*(T2)
(2) *SizeOf*(T1) = *SizeOf*(T2)
(3) *SizeOf*(T1) > *SizeOf*(T2)
- ANYTHING GOES -

Pascal/Modcal Special Case Compatibilities
(1) All Reference Types are Type Compatible with NIL
(2) All Set Types are Type Compatible with []

Appendix D: MODCAL Reserved Words

1. ANSI Standard Pascal Reserved Words

AND	FUNCTION	PROGRAM
ARRAY	GOTO	RECORD
BEGIN	IF	REPEAT
CASE	IN	SET
CONST	LABEL	THEN
DIV	MOD	TO
DO	NIL	TYPE
DOWNTO	NOT	UNTIL
ELSE	OF	VAR
END	OR	WHILE
FILE	PACKED	WITH
FOR	PROCEDURE	

2. HP Standard Pascal Additions

OTHERWISE

3. HP Standard MODCAL Additions

DEFINITION	IMPORT	READONLY
EXPORT	MODULE	RECOVER
HIDDEN	QUALIFIED	TRY
IMPLEMENT		

4. BCG Experimental MODCAL Additions

ANYVAR	CRUNCHED	OFFSET
OPTION	WITHOBJECT	